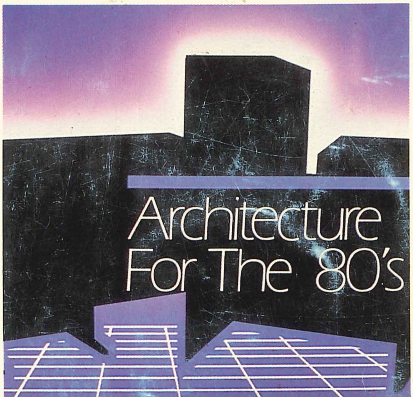
VAX



ARCHITECTURE HANDBOOK

digital

VAX

ARCHITECTURE HANDBOOK

digital

DIGITAL Facility, Boylston, Massachusetts

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.

VAX

ARCHITECTURE HANDBOOK

digital

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

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

This handbook was designed, produced, and typeset
by DIGITAL's New Products Marketing
using an in-house text-processing system.

CONTENTS

Preface

Designing the VAX Architecture

Several years ago, foreseeing the explosive growth of minicomputer uses, DIGITAL committed numerous corporate resources and personnel to the goal of creating a computer family for the 1980s and beyond. Extending and, at the same time, protecting our customers' enormous investment in PDP-11 computers and software lay at the heart of our efforts. But simultaneously we wanted to extend virtual address space to a degree that would eliminate the need for overlays and segmentation of programs. We also wanted to increase the capability and ease of use of our new computers, so that more and more applications could be handled by people with less and less specialized training. Along with this, we aimed for a computer that would serve a wider range of applications (OEM, laboratory realtime, distributed data processing, interactive, and so on) than any minicomputer from any other vendor.

What resulted from these efforts was the VAX family of 32-bit virtual memory minicomputers: high-speed, easy-to-use, highly dependable machines destined to satisfy our customers' needs for a decade and more. Central to these computers is the VAX architecture, an architecture that meets four critical design goals beyond that of maximal compatibility with the PDP-11s. We wanted high bit efficiency, and achieved it through an extensive collection of data types and addressing modes; we wanted a systematic, elegant instruction set, and achieved one with independence of operators, data types, and addressing modes, one that is easily exploited, particularly by high-level languages.

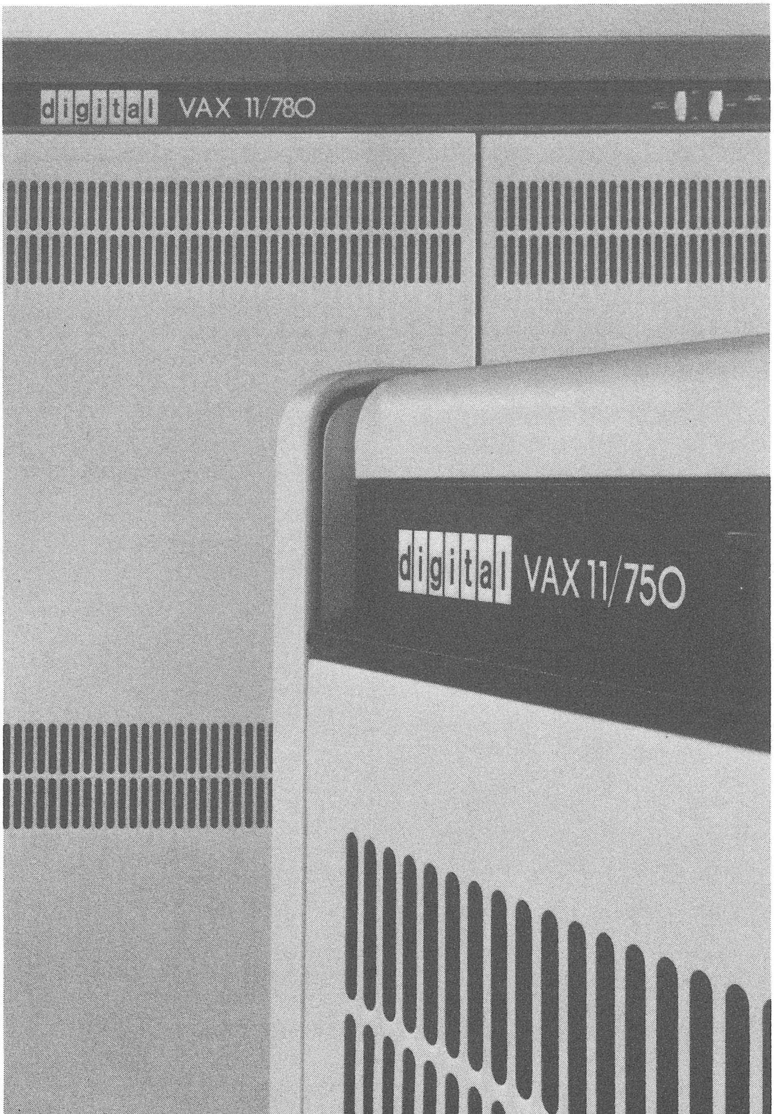
We wanted extensibility: new operators and data types may be added consistently with existing ones. And finally, we wanted a single architecture that would span and be suitable for all members of the VAX family, present and future.

In fulfilling these goals, DIGITAL engineers designed an architecture that provides you with numerous benefits above and beyond the high interactivity and friendliness our computers have always stood for:

- Programming time is minimized, so that programmer productivity is increased.
- Hardware peripherals are compatible with existing PDP-11 computers.

- Realtime response is significantly improved.
- The computer family and the program applications that run on it have a long life expectancy.
- Programs are completely transportable across VAX family members — providing adequate options and file spaces.
- It is easy to write programs that transport from VAXes to PDP-11s, and vice versa.
- There is much more code sharing on VAX machines than on any other computers in existence.
- System reliability is improved through architectural features.
- A single operating system, VAX/VMS, handles the full spectrum of application needs.
- And finally, the architecture creates an efficient, standard environment for all languages and the system to interface with each other ("anything can call anything").

We hope that the remainder of this Handbook will help you discover the power and elegance of the VAX architecture, and that you will see in it how we have satisfied our design goals.


digital VAX 11/780
digital VAX 11/750

CHAPTER 1
VAX: COMPUTERS FOR THE '80s

The next decade will witness ever-widening, perhaps unpredictable, demands upon computers and the computer industry. In finance, government, industry, and possibly even in the home, computers will serve expanding roles, solving problems, managing processes, or facilitating communication. DIGITAL has developed an innovative computer technology to confront the challenges of the 1980s, a technology that offers vast power and enormous flexibility for every kind of application. At the same time, we have held fast to the philosophy of affordability and easy use that made DIGITAL the minicomputer industry leader.

VAX is the name of our innovative computer family. In the short time since the introduction of the first family member, VAX has proven itself as one of the friendliest, and at the same time, most powerful, computer designs available. Its continual enhancement with new processors, peripherals, high-level languages, and other sophisticated software, positions VAX as the computer family able to meet the needs of the future.

Virtual Address Space

The letters *VAX* suggest the premier feature of VAX computers—Virtual Address eXtension. In a VAX computer, bytes of information are located with a 32-bit address. This means effectively that the computer can recognize more than four billion addresses, a vast amount in minicomputer and programmer's terms. The remarkable thing about this giant "address space" is the it is *virtual*: the (physical) main memory of the computer need not be anywhere near as large as four billion bytes for the machine actually to process data whose addresses are scattered through the address space. In fact, what happens is that a sophisticated scheme called "memory management" allows programmers to operate as if a big part of the virtual address space were really available to them, and then it handles all the details of storing programs and subsequently bringing them into main memory where they are processed.

From the programmer's point of view, the bottom two billion bytes of virtual address space can be used for programs, and he or she need never worry about complicated techniques of overlaying or segmenting to squeeze the program into a smaller address range. Logic built right into the VAX computers quickly translates all the programmer's virtual addresses into physical addresses, stores the programs and data in convenient locations (disks or main memory) and brings into main memory whatever parts of the program or data are needed at any instant.

Another aspect of memory management is the rapid switching of "contexts." VAX is a high-powered multiprocessor: many programs and many programmers can use it simultaneously, each appearing to own unique control of the processor. Actually, the computer is processing the programs—or pieces of them—one at a time, and switching into and out of main memory the "context" (loosely speaking, the environment) of many programs. A switched-in context allows a program to run; a switched-out context makes the program wait for the central processor. Consequently, numerous different activities could be occurring on a VAX computer at any one time: a data acquisition procedure, a long computation project, an editing session, an interprocess communication; and the context switching takes place so swiftly that each user would feel like the only user.

Scientific, industrial, commercial, and educational market users have already put the original VAX model through its paces in numerous situations: realtime, computational, program development. In the upcoming decade we will see a wide range of new usages handled by VAX computers.

At the heart of the VAX computer family is its architecture. For our purposes, architecture is the collection of attributes common to all family members, attributes that guarantee that all software runs without change on all family members. Particularly pertinent are the instruction set, the memory management algorithms, and certain other aspects of the design that help define contexts and processes.

A distinction should be made between the architecture and the implementation of that architecture. For example, the architecture of the typewriter is essentially fixed: it is the keyboard layout; knowing the alphabet and punctuation systems, any typist can make it work, can "process" jobs. Each manufacturer may, however, implement that architecture in individual ways. Some may have striking print keys, some may have typing head balls; some may have a blue keyboard, some black. In addition, the builder could trade off one feature against another: a lighter touch vs. the capability to make numerous carbons. Nevertheless, all machines still serve the essential function, typing.

Similarly with computer architecture. Each processor in the family may bear slightly differing implementations and tradeoffs; yet all will fulfill the core of requirements put on the machine by the designers, and all will deliver the same service to the users. Once having learned the instruction set, for example, a programmer is ensured that exactly the same instruction will perform precisely the same operation on each processor in the VAX family.

VAX architecture is appropriate over a variety of system costs, performance and application needs. Therefore, a huge range of user requirements can be met, at a lower cost, since the price of supporting many different architectures is eliminated.

Two very important aspects of the VAX architecture are its power and its connection to another major DIGITAL computer family, the PDP-11.

The most obvious manifestation of the VAX architecture is the instruction set. Over three hundred instructions give the assembly level programmer extensive control of computer operation. Each instruction has a mnemonic, a shorthand name that suggests its job. (Obvious ones are ADD, DIV, MOV, and PUSH). *Orthogonality* (i.e. independence) is incorporated into the instruction set. That is, the operation being performed (e.g., ADD), the type of data used (e.g., longword), and the method of addressing (e.g., autodecrement) can all be considered independently by the compiler. This makes for faster, more efficient, and easier to implement compilers.

In addition, each instruction operates on its "natural" number of operands, from zero up to as many as is appropriate. Also, some recurrent operations from high-level languages are engineered into the hardware, so that a single instruction can handle them. The FORTRAN DO loop and three-operand addition (A = B + C) are examples of operations that can be handled by a single VAX instruction. Finally, there is no forced alignment on longword boundaries: as required by many languages, data items bigger than a byte can still reside on any byte boundary.

The architecture also includes instructions to make various applications and operating system codes more efficient. There are, in this group, hardware support of queues, easy access to variable length bit fields, and simple instructions to save or restore a program context.

Because DIGITAL foresaw the possibility of more and more applications, the architecture is extendible. The instruction set can be expanded efficiently to include new data types and operators in a way that consistently matches all the ones that already exist. Enormous flexibility is assured this way, since what exists now does not significantly constrain what may be added in the future.

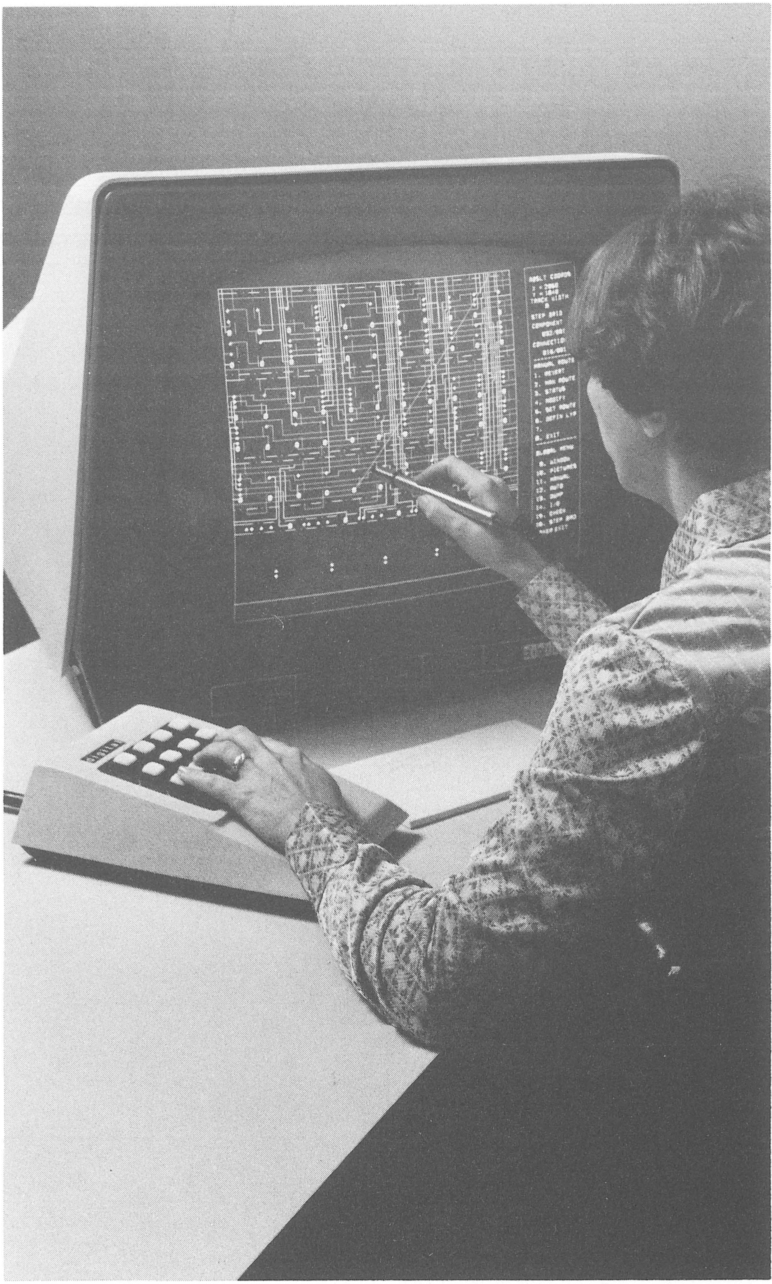
We use the word *compatibility* to designate VAX's connection to the PDP-11 family of minicomputers from DIGITAL. Customers have a large investment in the PDP-11 computers and software. To protect that investment, and to simplify the procedures by which programmers and programs can move back and forth between VAXs and PDP-11s, DIGITAL made sure that VAX would accept, with minimal conversion, most types of PDP-11 programs. Conversely, the VAX offers an excellent host development environment for applications that will eventually run on PDP-11 computers. Naturally, there are some restrictions, but in many cases, simple recompilation of programs is all that is required to carry a PDP-11 program to the VAX. VAX even has a compatibility mode at the hardware level, so that many PDP-11 programs can run unchanged on it. Compatibility mode may run along with "native" mode programs in a VAX multiprocessing environment.

The Architecture Handbook

This Handbook is part of the VAX Handbook Set. It is the most deeply technical of the three Handbooks in the set, and should probably be read by people with some computer familiarity. In it you will get a thorough view of the instruction set, of memory management, of process structure, and of PDP-11 compatibility mode. A companion volume, the VAX Software Handbook, describes in more generic terms the VAX/VMS operating system, optional languages, software routines, and system services. And the VAX Hardware Handbook gives a complete view of the processors in the VAX family.

Note that this Handbook uses a consistent set of notational conventions. You will find their descriptions conveniently grouped together in Appendix A1.

We hope that the Handbooks answer most of your questions about the VAX family, the computer architecture of the 1980s, and the huge selection of software available. If you have more questions, your DIGITAL Sales Representative will be happy to help you.


TRACK WIDTH
STEP GRID
COMPONENT
CONNECTIO
MANUAL ROUTE
1. REVERT
2. MAN ROUTE
3. STATUS
4. MODIFY
5. SET ROUTE
8. EXIT
15. STEP GRID

CHAPTER 2
VAX FAMILY ARCHITECTURE OVERVIEW

INTRODUCTION

The term "VAX Family Architecture" is most often abbreviated to "VAX Architecture." What is meant are the attributes of the system as seen by an assembly language programmer; that is, the conceptual structure and functional behavior (as distinct from the organization of the data flow and controls, the logical design, and the physical implementation). The primary advantage of a common family architecture is the ability to execute your software on any VAX family member.

For convenience, we may think of the architecture as divided into two major areas: application programmer (or user) aspects and system programmer aspects. Those attributes of the VAX family are part of the **user architecture** are:

- Four Gbyte virtual address space
- Data types
- Instruction formats
- Addressing modes
- Processor Status Word (low word of Processor Status Longword)
- User mode instructions in the native mode instruction set
- Compatibility mode instruction set
- User-visible aspects of exception handling

Those attributes which are part of the **system programmer architecture** are:

- Privileged instructions
- High word of the Processor Status Longword
- Process structure
- Memory management
- Interrupt structure and exception handling

PROCESS VIRTUAL ADDRESS SPACE

Note: If terms unfamiliar to you appear in subsequent sections, please refer to the Glossary and the Index at the end of the Handbook.

Most data are located in memory using the address of an 8-bit byte. Thirty-two-bit virtual addresses identify the byte locations. Such addresses are called *virtual* because they are not the real addresses for physical memory locations. Rather, they are translated into real addresses by the processor under operating system control.

A virtual address, unlike a physical memory address, is not a unique address of a location in memory. For example, two programs using the same virtual address might refer to two different physical memory locations; conversely, two programs could 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**. It can be viewed as an array of byte "locations" labelled from 0 to $2^{32} - 1$. This space is divided into sets of virtual addresses designated for certain uses: those used by processes constitute half of the total virtual address space, and are collectively designated as **process space**. Addresses in the remaining half of virtual address space refer to locations maintained and protected by the operating system, and are collectively designated as **system space**.

DATA TYPES

The data type of an instruction operand identifies how many bits of storage should be considered as a unit and what is to be the interpretation of that unit. This is important because, as you will see in later sections, identical bit patterns can be interpreted as very different data items; and, likewise, different bit patterns may be used to represent the same datum.

The processor's native instruction set recognizes six primary data types: integer and floating, character string, packed decimal, numeric string, and variable length bit field. For each of these data types, the selection of operation immediately informs the processor of the size and interpretation of the data, so that the processor can then manipulate the bit field as a function of user-defined field size and relative position from a given byte address.

There are several variations on the six primary data types. Table 2-1 provides a summary of all the data types available, and Figure 2-1 illustrates some of them graphically.

Integer data are stored as binary values. An integer can be stored in a byte, word, longword, quadword, or, in some cases, in an octaword. A byte is eight bits, a word is two bytes, a longword is four bytes, a quadword is eight bytes, and an octaword is sixteen bytes. The processor can interpret an integer as either a signed value (sign is determined by the high-order bit) or an unsigned value.

Table 2-1 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 $2^{64}-1$
Octaword	128 bits	-2^{127} to $+2^{127}-1$	0 to $+2^{128}-1$
Floating Point			
F floating	32 bits	approximately seven decimal digits precision	
D floating	64 bits	approximately sixteen decimal digits precision	
G floating	64 bits	approximately fifteen decimal digits precision	
H floating	128 bits	approximately thirty-three 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	
Numeric String	0 to 31 bytes (DIGITS)	$-10^{31}-1$ to $+10^{31}-1$	
Queue	≥ 2 longwords/queue entry	0 through 2 billion entries	

Floating point values are stored using a signed exponent and a binary fraction. Four types of floating point data formats are provided. The two PDP-11 compatible formats (F_floating and D_floating) are standard on all VAX family processors. Two extended range formats (G_floating and H_floating) are available as options on VAX family processors. F_floating and D_floating are four and eight bytes long, respectively. F_floating data yield approximately 7 decimal digits of precision, while D_floating yields approximately 16 decimal digits of precision. G_floating is also eight bytes in length, but because of the different arrangement of the fraction and exponent parts, its precision is approximately 15 decimal digits. H_floating is 16 bytes in length with a 15-bit exponent and 113-bit fraction. As a result, its precision is approximately 33 decimal digits.

Character data are simply a string of bytes containing any binary data, for example, ASCII codes. The first character in the string is stored in the first byte, the second character is stored in the second byte, and so on. In particular, a character string that contains ASCII codes for decimal digits is called a numeric string.

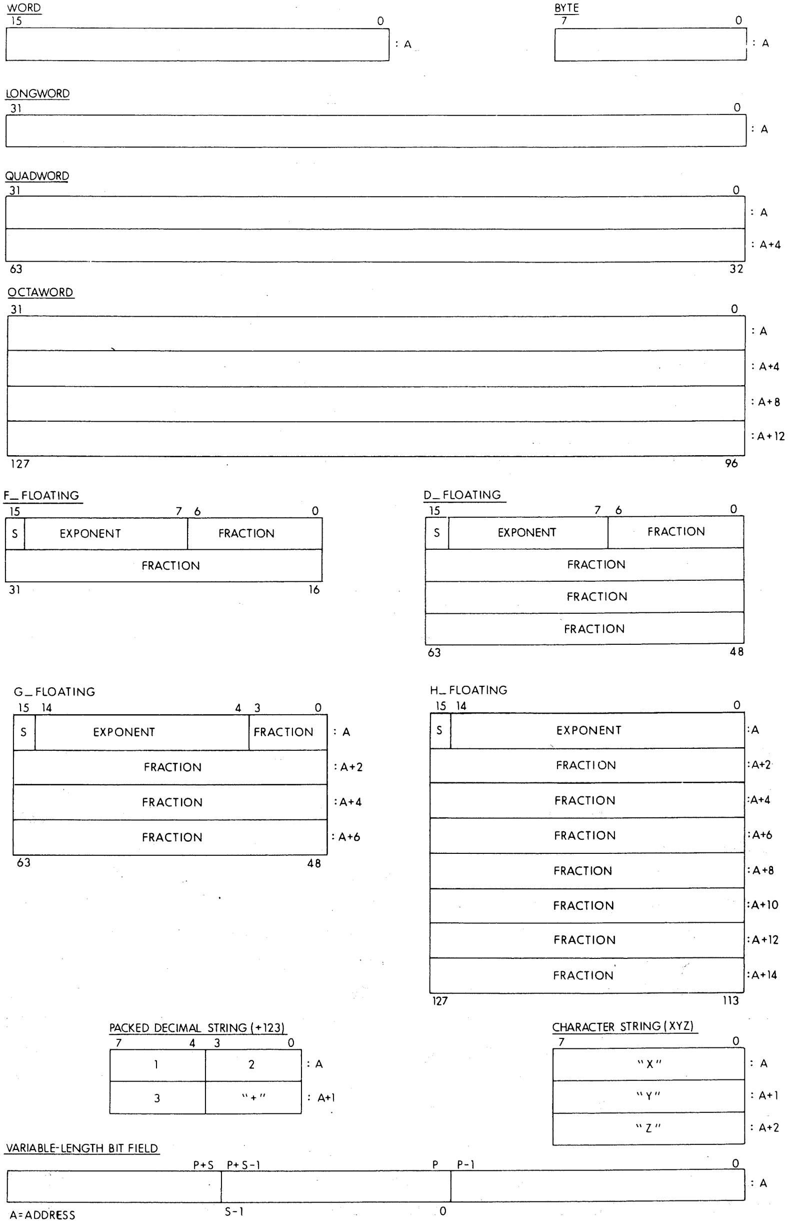


Figure 2-1 Data Type Representations

Numeric String Data are representations of fixed quantities, using one byte of the string for each decimal digit. As detailed in Chapter 4, the variety of external data arrangements demands a variety of matching numeric string forms; particularly, it is necessary to know whether the sign of the number appears in the first byte or as part of the last byte.

Packed decimal data are stored in a string of bytes. Each byte is divided into two 4-bit nibbles with one decimal digit stored in each nibble. The first, or most significant digit is stored in the high-order nibble of the first byte, the second digit is stored in the low-order nibble of the first byte, the third digit is stored in the high-order nibble of the second byte, and so on. The sign of the number is stored in the low-order nibble of the last byte of the string.

Queue data are held in circular, doubly-linked lists (that is, each entry is accompanied by two longwords, one of which tells the location of the succeeding entry, one of which specifies the location of the preceeding entry). Two kinds of queue data exist: absolute queues use absolute addresses; relative queues use relative addresses. Chapter 12 includes the Queue instructions, and more detail.

Variable length bit field data are small integers packed together in a larger data structure. Basically, they are used to increase memory efficiency.

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. It 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.

GENERAL REGISTERS AND ADDRESSING MODES

Within the processor there are locations called **general registers** that can be used for temporary data storage and addressing. Sixteen 32-bit general registers are available for use with the native instruction set, though some have special significance. For example, 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 method by which an operand location is specified is called the **operand addressing mode**. VAX processors offer a variety of addressing modes and addressing mode optimizations: one addressing mode locates an operand in a register; several other addressing modes locate an operand in memory by using a register to: 1) point to the operand, 2) point to a table of operands, or 3) point to a table of operand addresses.

There are also addressing modes that are indexed modifications of the addressing modes which locate an operand in memory. Finally, there are addressing modes that identify the location of the operand in the instruction stream, including one for constant data and one for branch instruction addresses. The VAX addressing modes are briefly summarized here in Table 2-2, but they are explained in greater details and with examples in Chapter 5.

Table 2-2 Addressing Modes

Mode	Symbol	
Literal (Immediate)	$\left\{\begin{matrix}S\uparrow\\I\uparrow\end{matrix}\right\}$ #*constant*	
Register	R*n*	
Register Deferred	(R*n*)	Indexed [Rx]
Autodecrement	-(R*n*)	
Autoincrement	(R*n*) +	
Autoincrement Deferred	@ (R*n*) +	
(Absolute)	@ #*address*	
Displacement	$\left\{\begin{matrix}B\uparrow\\W\uparrow\\L\uparrow\end{matrix}\right\}$ *displacement* (R*n*) *address*	
Displacement Deferred	@$\left\{\begin{matrix}B\uparrow\\W\uparrow\\L\uparrow\end{matrix}\right\}$ *displacement* (R*n*) *address*	

n = 0 through 15
x = 0 through 14

PROCESSING CONCEPTS FOR SYSTEM PROGRAMMING

The VAX processor is specifically designed to support a high-performance multiprogramming environment. The major advantage of a multiprogramming system is its ability to utilize most efficiently those resources of the computer that are being shared by several executing environments. For example, multiprogramming enables the execution of many applications and the interactive development of applications programs simultaneously. Hardware characteristics that support multiprogramming are:

- Rapid context switching
- Priority dispatching
- Virtual addressing and memory management

As a multiprogramming system, VAX not only gives the power to share the processor among processes, but also protects processes from one another while enabling them to communicate with each other and share code and data.

Context Switching

In a multiprogramming environment, several individual streams of code can be ready to execute at any one time. Instead of allowing each stream to execute to completion serially (as in a batch-only stream), the operating system can intervene and switch among them. In VAX family computers, the hardware establishes an environment for rapid switching. Switching occurs to increase the efficiency of the computer by exploiting its resources in a balanced fashion, and to allow the intervention of processes or events that require priority treatment.

The stream of code executing at any one time is determined by its **hardware context**, information loaded in the processor's registers that identifies:

- Location of the stream's instructions and data
- Which instruction to execute next
- Processor status during execution

A **process** is a stream of instructions and data defined by a hardware context. Each process has a unique identification in the stream. 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.

Priority Dispatching

While running in the context of one process, the processor executes instructions and controls data flow to and from peripherals and main memory. To share processor, memory, and peripheral resources among many processes, the processor has two arbitration mechanisms that support high-performance multiprogramming: exceptions and interrupts. Exceptions are events that occur synchronously (predictably) with respect to the execution of a particular stream of instructions, 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 systemwide. Process-local changes occur as the result of a user software error or when user software calls operating system services. They are handled through the processor's exception detection mechanism and the operating system's exception dispatcher.

Systemwide 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. (Systemwide changes in flow may also occur as the result of severe hardware errors, in which case they are handled either as special exceptions or high-priority interrupts.)

Systemwide changes in flow take priority over process-local ones. Furthermore, the processor uses a priority system for servicing interrupts. Each kind of interrupt is assigned a priority, and the processor responds to the highest priority interrupt pending. For example, interrupts from the high-speed disk devices take precedence over interrupts from low-speed devices.

The processor sevices interrupts between instructions, or at well-defined points during the execution of long, iterative instructions. When it acknowledges an interrupt, it switches rapidly to a special systemwide context to enable the operating system to service the interrupt. Systemwide changes in the flow of execution are handled in such a way as to be totally transparent to individual processes.

SYSTEM PROGRAMMING ENVIRONMENT

Within the context of any 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 are the Processor Status Word available to the user process (see below). The high-order 16 bits provide privileged control of the system.

PSL fields can be grouped together by functions that control:

- The access mode of the current instruction
- The instruction set the processor is executing
- Interrupt processing

Processor Access Modes

In a high-performance multiprogramming system, the processor must provide the basis for protection and sharing among all the processes competing for the system's resources. The basis for protection in this system is the processor's access mode. The access mode is responsible for determining both the:

- Instruction execution privileges: what instructions the processor will execute, and the
- Memory access privileges: which locations in memory the current instruction can access.

At any one time, the processor is executing code either in a particular process, or in the systemwide 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 execution time in user mode in the context of one process or another. When user software needs the more privileged services of the operating system, whether for acquisition of a resource, for I/O processing, or for information, it calls those services, and the processor executes them, either in the process's access mode or a more privileged one.

Only in kernel mode will the processor execute an instruction that halts the processor, loads and saves process context, or accesses the internal processor registers controlling memory management, interrupt processing, the console, or the processor clock.

The ability to execute code in one of the more privileged modes is granted by the *system manager* and controlled by the operating system. 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.

Protected and Privileged Instructions

The processor provides three types of instructions that enable user-mode software to obtain privileged services without jeopardizing the integrity of the operating system. They are:

- Change Mode instructions
- PROBE instructions
- 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. A Change Mode instruction is simply a special trap instruction that can be thought of as an operating system service call instruction. Ultimately the system manager grants the privilege to write any code that handles Change Mode traps to more privileged access modes.

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 makes the operating system provide services that execute in privileged modes to less privileged callers while still preventing 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. REI restores the Program Counter and the processor state to resume the process at the point where it was interrupted.

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. The operating system's context switching procedure identifies the location of the hardware context to be loaded by updating an internal processor register.

MEMORY MANAGEMENT

The processor is responsible for enforcing memory protection between access modes, but that 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 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 physical address. (A physical address on the other hand, is the address exchanged between the processor, memory, and the peripheral adapters. Typically, the physical address is transparent to the programmer, who deals with virtual addresses.)

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 0 and continues to byte 511. The next page begins at byte 512 and continues to byte 1023, and so forth. If we listed the first three pages of virtual address space, their addresses in both decimal and hexadecimal notation are:

PAGE	ADDRESS$_{10}$	ADDRESS$_{16}$
0	0000-0511	0000-01FF
1	0512-1023	0200-03FF
2	1024-1535	0400-05FF

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.

The processor has three pairs of page mapping registers, one pair for each of the three major regions of virtual address space that are actively used. The operating system's memory management software loads the pairs of registers with base addresses and lengths of data structures called **page tables**, which provide the mapping information for each virtual page in the system. Thus, there is one page table for each of the three regions.

A page table is a virtually contiguous array of page table entries, each of which 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.

All process page tables have virtual addresses in the system region of virtual address space, but the system region page table itself is located by its address in physical memory. That is, the system region page table base register contains the physical address of the page table base, while the process page table base registers contain the virtual addresses of their page table bases.

There are two advantages to using a virtual address as the base address of a per-process page table. The first is that all page tables do not have to reside in physical memory. The system region page table is the only page table that needs to be resident in physical memory. The process page tables can reside on disk; that is, they can themselves be paged and swapped as necessary.

The second advantage is that the operating system's memory management software can allocate process page tables dynamically, because they do not need to be mapped into contiguous physical pages. And although the system region page table must be mapped into contiguous physical pages, this requirement does not restrict physical memory allocation. The region is shared among processes, and therefore does not require redefinition from context to context.

Because of the complexity of memory management, specialists may want to study it in greater detail, particularly as those details are presented in Chapters 6 and 7.

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, where the exception and interrupt vectors are contained. The processor locates each vector by using a specific offset into 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.

To handle interrupt requests, the processor enters a special systemwide context. In the systemwide context, the processor executes in kernel mode using a special data structure 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 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.

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. Software reads the registers to obtain the controller status. The driver controls interrupt enabling and disabling on the set of controllers for which it is responsible. If 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 it is a pointer to another data structure, the **hardware process control block**, which contains the hardware process context, that is, 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 they are delivered to the proper access mode for the process.

STACKS, SUBROUTINES, AND PROCEDURES

A stack is an array of consecutively addressed data items referenced on a last-in, first-out (LIFO) 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 the program's address space and can use 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 refers to its stack data structure, called the **user stack**, through a general register designated as the Stack Pointer (SP). When you run a program image, the operating system automatically provides the address of the area designated for the user stack.

A stack is an extremely powerful data structure because it can be used to pass arguments to routines efficiently. In particular, the stack structure enables the coding of reentrant routines because the processor can handle routine linkages automatically using the Stack Pointer. Routines can also be recursive: 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 automatically saves and restores the contents of registers to be preserved across procedure calls, and it provides two methods for passing argument lists to called procedures: by passing the arguments on the stack, or by passing addresses of arguments elsewhere in memory. The processor also constructs a "journal" 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 the procedures leave data on the stack. In addition, user and operating system software can "unwind" the stack frame to trace back through nested calls to handle errors or debug programs.

INSTRUCTION FORMAT

A native-mode instruction may start on any byte boundary. VAX's variable-length instruction format not only makes code more compact, but it also guarantees that the instruction set can be easily extended. Opcodes for the operations are single or double bytes followed by zero to six operand specifiers, depending on the instruction. An operand specifier can be one to ten bytes long, depending on the addressing mode (see Chapter 5). Figure 2-2 illustrates the autodecrement mode Move Long instruction as a string of bytes starting with the opcode followed by two operand specifiers. In this example, the assumed starting location is 00003000. When the processor completes the execution of an instruction, the Program Counter contains the address of the first byte of the next instruction. The Program Counter operation is totally transparent to the programmer.

MACHINE CODE: (ASSUMED STARTING LOCATION 00003000)

00003000	D0	OPCODE FOR MOVE LONG INSTRUCTION
00003001	73	AUTODECREMENT MODE, REGISTER R3
00003002	54	REGISTER MODE, REGISTER R4

Figure 2-2 Autodecrement Move Long Instruction

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, which is also called *immediate mode*, is used to specify in-line constants other than those available with literal mode addressing. Autoincrement deferred mode using the Program Counter, 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 coding of 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.

COMPATIBILITY MODE

Under control of the operating system, the processor can execute PDP-11 instruction streams within the context of any process. When executing in compatibility mode, the processor interprets the instruction stream of the current process as a subset of PDP-11 code. The subset does not include floating point hardware or privileged instructions.

In general, compatibility mode enables the operating system to provide an environment for executing most user-mode programs written for a PDP-11. The processor expects all compatibility mode software to rely on the services of the native operating system for I/O processing, interrupt and exception handling, and memory management. There are some restrictions, however, on the environment that the native operating system can provide a PDP-11 program (see Chapter 13 for more detail).

FOR MORE INFORMATION ON VAX ARCHITECTURE

Expanded information on the VAX Architecture is available in the subsequent chapters of this handbook, and the VAX-11 Hardware Handbook. Your DIGITAL software support specialist and sales representative are also good sources of further information.

CHAPTER 3

SYSTEM ARCHITECTURAL CHARACTERISTICS

INTRODUCTION

The common VAX architecture defines a consistent *functional* behavior seen by programmers regardless of which VAX family processor they use. Areas of interaction that, from the programmer's viewpoint, display this processor independence include: data sharing and synchronization, restartability, interrupts and errors, and 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 family architecture, implicit sharing is transparent to the programmer. The memory system is implemented such that the basis of access for independent modification is the byte. Notice that this does not imply a maximum reference size of one byte, but only that independent modifying accesses to adjacent bytes produce the same results regardless of the order of execution. For example, locations 0 and 1 contain the values 5 and 6 respectively. One process executes INCB 0 (increment by 1 the byte at location 0) and another executes INCB 1. Then, regardless of the order of execution (including effectively simultaneous execution) the final contents must be 6 and 7.

Access to explicitly shared data that may be written must be synchronized. Before accessing shared writable data, the programmer must acquire control of the data structure. Seven instructions are provided to permit interlocked access to a control variable. The Branch on Bit Set and Set, Interlocked (BBSSI) and Branch on Bit Clear and Clear, Interlocked (BBCCI) instructions use hardware-provided primitive operations to make a read and a subsequent write reference to a single bit within a single byte in an interlocked sequence. The Add Aligned Word, Interlocked (ADAWI) instruction uses a hardware-provided primitive operation to make a read and a subsequent write operation to a single aligned word in an interlocked sequence to allow counters to be maintained without other interlocks. The Insert into Queue Head, Interlocked (INSQHI), Insert into Queue Tail, Interlocked (INSQTI), and the analogous Remove instructions, REMQHI and REMQTI, use a primitive operation provided by hardware to make a series of aligned longword reads and writes in an interlocked method to allow queues to be maintained without other interlocks. Use of the hardware primitives guarantees that no read operation within the synchronizing part of these instructions can occur between the synchronized reads and the writes of the instructions. Such instructions are implemented so that no faults will cause the data structure to be locked for an extended period. On the processor, only interlocking instructions are locked out by the interlock.

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

CACHE

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

1. Program writes to memory, followed by a peripheral output transfer, output the updated value.
2. A peripheral input transfer followed by a program reading of memory reads the input value.
3. A write or modify followed by a HALT on one processor, followed by a read or modify on another processor, reads the updated value. (Note that this only applies to customer-designed multiprocessor systems.)
4. A write or modify followed by a power failure, followed by restoration of power, followed by a read or modify, reads the updated value. This only occurs if the duration of the power failure does not exceed the maximum nonvolatile period of the main memory or if the contents of memory were protected by optional battery backup.
5. In customer-designed multiprocessor systems, access to variables shared between processors is interlocked by software executing BBSSI, BBCCI, ADAWI, INSQHI, INSQTI, REMQHI, or REMQTI instructions. In particular, the writer must execute an interlocking instruction after the write to release the interlock and the reader must execute a successful matching interlock instruction before the read.
6. Accesses to I/O registers are not cached.

RESTARTABILITY

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

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

Instruction operands that are peripheral device registers having access side effects may produce unpredictable results due to instruction restarting after faults (including page faults) or interrupts. To ensure no interrupts, the programmer avoids operand specifier addressing modes 9, 11, 13, and 15, and the indexed forms of these modes. (Symbolically, @(Rn)+, @B↑D(Rn), @W↑D(Rn), and @L↑D(Rn), and these indexed.) The hardware, however, may allow interrupts for these modes in order to minimize interrupt latency.

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

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

INTERRUPTS AND ERRORS

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

So that software can operate deterministically, any instruction changing the processor priority (IPL) to enable a pending interrupt, allows the interrupt to occur before the next instruction that would normally have been executed.

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

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

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

I/O STRUCTURE

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. Use of general instructions permits all the virtual address mapping and protection mechanisms described in the Memory Management chapter to be used when referencing I/O registers.

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

Constraints on I/O Registers

I/O registers satisfy the following simple rules and constraints:

1. All registers are aligned on natural boundaries, and the physical address of an I/O register will always be an integral multiple of the register size in bytes (which must be a power of 2).
2. References using a length attribute other than the length of the register and/or an unaligned reference may produce unpredictable results. For example, a byte reference to a word-length register will not necessarily respond by supplying or modifying the byte addressed.
3. In peripheral devices, error and status bits that may be asynchronously set by the device are usually cleared by software writing a one to these bits, and are not affected by writing a zero. This is to prevent clearing bits that may be asynchronously set between reading and writing a register.
4. Only byte and word references of a read-modify-write 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, and this includes the BBSSI and BBCCI instructions.

5. String, quadword, octaword, F_floating, D_floating, G_floating, H_floating, and field references in the I/O space result in undefined behavior.

The information on VAX architectural characteristics will assist sophisticated users to modify or enhance the capabilities of VAX systems. It should also be of help to designers who are configuring their own multiprocessor systems. For additional details, please consult your DIGITAL Sales Representative or Software Specialist.

CHAPTER 4
DATA REPRESENTATION

INTRODUCTION

The VAX instruction set can use a wide range of data types, all of which can be separated into categories according to the groups of instructions that operate on them. They are:

- Integer and Floating Data Types
- Character String Data Types
- Numeric String Data Types
- Packed Decimal Data Types
- Queue Data Types
- Variable Length Bit Field Data Types
- Special Table Data Types

Figure 4-1 summarizes the VAX data types. Many of these data types can be further characterized by both size and format flexibility.

INTEGER AND FLOATING DATA TYPES

In the following discussion of integer and floating data types, the address of the datum in memory is the address of the byte of the datum with the lowest address. When depicted, this lowest byte is shown on the right and in discussions this is what is meant when the word "right" is used.

Integer Data

VAX supports integer data types of 8-, 16-, 32-, 64-, and 128-bit sizes. These are termed **byte, word, longword, quadword,** and **octaword** integers, respectively. The integer data types are stored in memory in a binary format which can be treated as either signed or unsigned quantities. As unsigned quantities, integers extend upward from 0. As signed quantities, the integers are represented in 2's complement form. This means that positive numbers have a zero most significant bit (MSB) and the representation of a negative number is one greater than the bit-by-bit complement of its positive counterpart. Thus, the MSB is always zero for positive values and one for negative values.

- Byte

A byte is eight contiguous bits starting on an addressable byte boundary or located in a register, Rn<7:0>. The bits are numbered from the right 0 through 7.

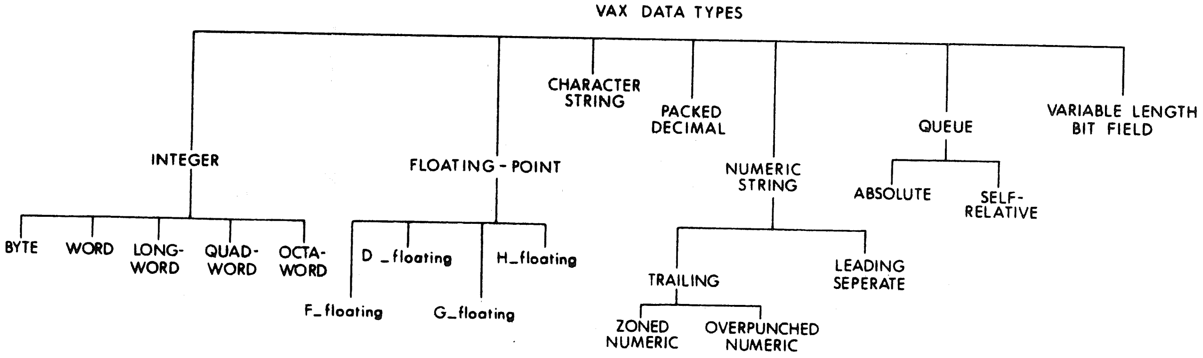


Figure 4-1 VAX Data Types

The byte is specified by its address A. When interpreted as a signed quantity, a byte is a 2's complement integer with bits increasing in significance from 0 through 6, and with bit 7 designating the sign.

- Word

A word, two contiguous bytes, starts on an arbitrary byte boundary or is located in a register Rn<15:0>. The bits are numbered from the right 0 through 15.

Words, longwords, quadwords, and octawords are specified by their address A, the address of the byte containing bit 0. When interpreted as a signed quantity, a word is a 2's complement integer with bits increasing in significance from 0 through 14, and with bit 15 designating the sign.

- Longword

A longword is four contiguous bytes starting on an arbitrary byte boundary or located in a register Rn<31:0>. The bits are numbered from the right 0 through 31.

When interpreted as a signed quantity, a longword is a 2's complement integer with bits increasing in significance from 0 through 30, and with bit 31 designating the sign.

- Quadword

A quadword is eight contiguous bytes starting on an arbitrary byte boundary or in two consecutive registers, R[n+1]'R[n]. The bits are numbered from the right 0 through 63.

A quadword is specified by its address A, the address of the byte containing bit 0. When interpreted as a signed quantity, a quadword is a 2's complement integer with bits increasing in significance from 0 through 62.

- Octaword

An octaword is sixteen contiguous bytes starting on an arbitrary byte boundary. The bits are numbered from the right 0 through 127.

When interpreted arithmetically, an octaword is a 2's complement integer with bits of increasing significance going 0 through 126, and bit 127 the sign bit. The octaword data type is not yet fully supported by VAX instructions.

Floating Point Data

The floating point data types represent approximations to quantities using a scientific notation consisting of a sign, the exponent of a power of two, and a fraction between .5 (inclusive) and 1.0 (exclusive). VAX supports, in the instruction set, floating point data types of 32-, 64-, and 128-bit sizes. Essentially four floating point data types are available, two of 8 bytes in length (D_floating and G_floating) and one each of lengths 4 bytes and 16 bytes (F_floating and H_floating, respectively).

- F_floating (single-precision floating)

An F_floating datum, sometimes called just "floating" or "single-precision floating," is four contiguous bytes starting on an arbitrary byte boundary or in register n. The bits are labelled from the right 0 through 31.

The form of an F_floating datum is sign magnitude with bit 15 the sign bit, bits 14:7 an excess 128 binary exponent, and bits 6:0 and 31:16 a normalized 24-bit fraction with the redundant most significant fraction bit not represented. Within the fraction, bits increase in significance from 16 through 31 and 0 through 6. The 8-bit exponent field encodes the values 0 through 255. An exponent value of 0 together with a sign bit of 0, indicates that the F_floating datum has a value of 0. Exponent values of 1 through 255 indicates true binary exponents of −127 through +127. An exponent value of 0, together with a sign bit of 1, is taken as reserved. (Floating point instructions processing a reserved operand take a reserved operand fault.) The magnitude of an F_floating datum is in the approximate range $.29*10^{-38}$ through $1.7*10^{38}$. The precision of an F_floating datum is approximately one part in 2^{23} (approximately 7 decimal digits.)

- D_floating (double-precision floating)

The D_floating datum sometimes referred to as "double floating" or "double-precision floating," is eight contiguous bytes starting on an arbitrary byte boundary or in two consecutive registers, R[n+1]'R[n]. The bits are labelled from the right 0 through 63.

The form of a D_floating datum is identical to the F_floating datum except for an additional 32 low significance fraction bits. Within the fraction, bits increase in significance from 48 through 63, 32 through 47, 16 through 31, and 0 through 6, as suggested by the widening arrow in Figure 4-2. The exponent conventions, and approximate range of values is the same for both D_floating and F_floating. The precision of a D_floating datum is approximately one part in 2^{55} (approximately 16 decimal digits).

- G_floating

A G_floating datum is eight contiguous bytes starting on an arbitrary byte boundary. The bits are labelled from the right 0 through 63.

The form of a G_floating datum is sign magnitude with bit 15 the sign

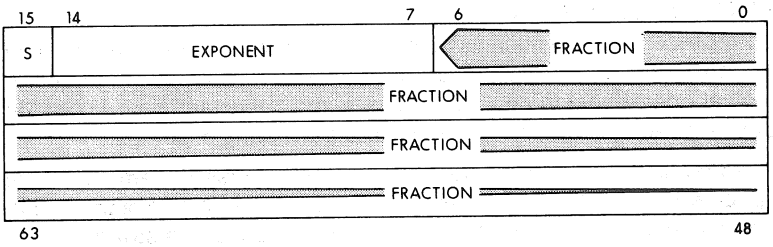


Figure 4-2 D_floating Format

bit, bits 14:4 an excess-1024 binary exponent, and bits 3:0 and 63:16 a normalized 53-bit fraction with the redundant most significant fraction bit not represented. Within the fraction, bits of increasing significance go 48 through 63, 32 through 47, 16 through 31, and 0 through 3. The 11-bit exponent field encodes the values 0 through 2047. An exponent value of 0 together with a sign bit of 0, is taken to indicate that the G_floating datum has a value of 0. Exponent values of 1 through 2047 indicate true binary exponents of −1023 through +1023. An exponent value of 0, together with a sign bit of 1, is taken as reserved. Floating point instructions processing a reserved operand take a reserved operand fault. The value of a G_floating datum is in the appropriate range $.56*10^{-308}$ through $.9*10^{308}$; the precision is approximately one part in 2^{52} (typically 15 decimal digits).

- H_floating

An H_floating datum is 16 contiguous bytes starting on an arbitrary byte boundary. The bits are labelled from the right 0 through 127.

The form of a H_floating datum is sign magnitude with bit 15 the sign bit, bits 14:0 an excess-16384 binary exponent, and bits 127:16 a normalized 113-bit fraction with the redundant most significant fraction bit not represented. Within the fraction, bits of increasing significance go 112 through 127, 96 through 111, 80 through 95, 64 through 79, 48 through 63, 32 through 47, and 16 through 31. The 15-bit exponent field encodes the values 0 through 32767. An exponent value of 0 together with a sign bit of 0 is taken to indicate that the H_floating datum has a value of 0. Exponent values of 1 through 32767 indicate true binary exponents of −16383 through +16383. An exponent value of 0, together with a sign bit of 1, is taken as reserved. Floating point instructions processing a reserved operand take a reserved operand fault. The value of an H_floating datum is in the approximate range $.84*10^{-4932}$ through $.59*10^{4932}$. The precision of a H_floating datum is approximately one part in 2^{112} (typically 33 decimal digits).

CHARACTER STRING DATA TYPE

To represent strings such as names, data records, or text, you can use the character string data type. Rather than performing arithmetic or logical operations on character strings, the important instructions do copying, searching, concatenating, and translating of strings.

A character string is a contiguous sequence of bytes in memory, and is specified by two attributes: the address A of the first byte of the string, and the length L of the string in bytes. The length L of a string is in the range 0 through 65,535. A string with length 0 is termed a null string; it contains no bytes and no memory is referenced; hence, the address need not be valid.

The format of a character string is illustrated in Figure 4-3.

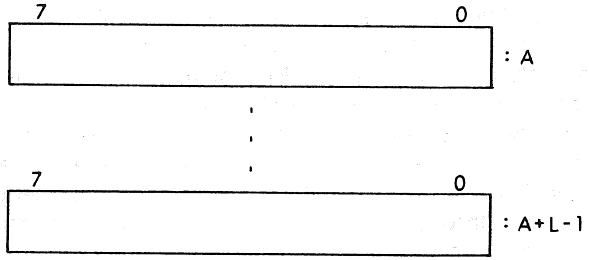


Figure 4-3 Character String Format

The address of a string specifies the first character of a string. In the following example, **XYZ** is represented as:

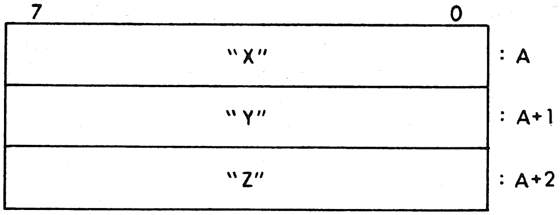


Figure 4-4 String **XYZ**

NUMERIC STRING DATA TYPES

Numeric string data types are used to represent fixed scaled quantities in forms close to their external representations. For programs that are input/output intensive rather than computation intensive, this presentation can be efficient. The decimal string form also provides greater precision than floating point and greater range than integer data types.

There are two forms of decimal data on VAX; the decimal string data types in which each decimal digit occupies one byte, and a more compact form (discussed later) in which two decimal digits are packed into one byte. These are termed **numeric** and **packed decimal strings**, respectively. Because the numeric string form must represent many external data arrangements exactly, it appears in several forms. The most significant distinguishing characteristic is whether the sign, if any, appears before the first digit or is superimposed on the final digit. The first is called the "leading separate numeric string," while the second is the "trailing numeric string."

Leading Separate Numeric String

A leading separate numeric string is a contiguous sequence of bytes in memory. It is specified by two attributes: the address A of the first byte (containing the sign character), and a length L that is the length of the string in digits and NOT the length of the string in bytes: the number of bytes in a leading separate numeric string is L+1. The address A of the string specifies the byte of the string containing the sign. Digits of decreasing significance are assigned to bytes of increasing addresses.

The sign of a separate leading numeric string is stored in a separate byte. Valid sign bytes are:

sign	decimal	hex	ASCII character
+	43	2B	+
+	32	20	<blank>
–	45	2D	–

The preferred representation for + is 2B, the ASCII +. All subsequent bytes contain an ASCII digit character:

digit	decimal	hex	ASCII character
0	48	30	0
1	49	31	1
2	50	32	2

digit	decimal	hex	ASCII character
3	51	33	3
4	52	34	4
5	53	35	5
6	54	36	6
7	55	37	7
8	56	38	8
9	57	39	9

The length L of a leading separate numeric string must be in the range 0 to 31 (0 to 31 digits). The value of a 0 length string is identically 0; it contains only the sign byte.

The following examples illustrate the representations of +**123** and −**123** in leading separate numeric string format.

+**123** is represented as:

ZONED FORMAT OR UNSIGNED

7 4	3 0	
3	1	: A
3	2	: A+1
3	3	: A+2

OVERPUNCH FORMAT

7 4	3 0	
3	1	: A
3	2	: A+1
4	3	: A+2

and −**123** is represented as:

ZONED FORMAT

7 4	3 0	
3	1	: A
3	2	: A+1
7	3	: A+2

OVERPUNCH FORMAT

7 4	3 0	
3	1	: A
3	2	: A+1
4	C	: A+2

Trailing Numeric String

A trailing numeric string is a contiguous sequence of bytes in memory. The string is specified by two attributes: the address A of the first byte (most significant digit) of the string, and the length L of the string in bytes.

Note that the address A of the string specifies the byte of the string containing the most significant digit. Digits of decreasing significance are assigned to increasing addresses.

All bytes of a trailing numeric string, except the least significant digit byte, must contain ASCII decimal digit characters (0-9). The representation for these "non-least significant" digits is:

digit	**decimal**	**hex**	**ASCII character**
0	48	30	0
1	49	31	1
2	50	32	2
3	51	33	3
4	52	34	4
5	53	35	5

digit	decimal	hex	ASCII character
6	54	36	6
7	55	37	7
8	56	38	8
9	57	39	9

The highest addressed byte of a trailing numeric string represents an encoding of both the least signficant digit and the sign of the numeric string. The VAX-11 numeric string instructions support any encoding; however there are three preferred encodings used by DIGITAL software. These are (1) unsigned numeric in which there is no sign and the least significant digit contains an ASCII decimal digit character, (2) zoned numeric, and (3) overpunched numeric. Because the overpunch format has been used by the compilers from numerous manufacturers over many years, and because various card encodings are used, several variations in overpunch format have evolved. Typically, these alternate forms are accepted on input. The normal form is generated on output of all operations. The valid representations of the digit and sign in each of the latter two formats is shown in Table 4-1.

Table 4-1 Representation of Least Significant Digit and Sign

Zoned Numeric Format				Overpunch Format			
	deci-		ASCII	deci-		ASCII char.	
digit	mal	hex	char.	mal	hex	norm	alt.
0	48	30	0	123	7B	{	0 [?
1	49	31	1	65	41	A	1
2	50	32	2	66	42	B	2
3	51	33	3	67	43	C	3
4	52	34	4	68	44	D	4
5	53	35	5	69	45	E	5
6	54	36	6	70	46	F	6
7	55	37	7	71	47	G	7
8	56	38	8	72	48	H	8
9	57	39	9	73	49	I	9
−0	112	70	p	125	7D	}	] : !
−1	113	71	q	74	4A	J	
−2	114	72	r	75	4B	K	
−3	115	73	s	76	4C	L	
−4	116	74	t	77	4D	M	

	Zoned Numeric Format			Overpunch Format			
digit	decimal	hex	ASCII char.	decimal	hex	ASCII char. norm	alt.
−5	117	75	u	78	4E	N	
−6	118	76	v	79	4F	O	
−7	119	77	w	80	50	P	
−8	120	78	x	81	51	Q	
−9	121	79	y	82	52	R	

The length L of a trailing numeric string must be in the range 0 to 31 (0 to 31 digits). The value of a 0 length string is identically 0; it contains no bytes and no memory is referenced; hence, the address need not be valid.

The following examples illustrate the representations of **123** and **−123** in trailing numeric string format.

Thus **123** is represented as:

7 4	3 0	
2	B	: A
3	1	: A+1
3	2	: A+2
3	3	: A+3

and −**123** is represented as:

7 4	3 0	
2	D	: A
3	1	: A + 1
3	2	:A + 2
3	3	:A +3

PACKED DECIMAL STRING

A packed decimal string is a contiguous sequence of bytes in memory. The address A and length L are sufficient to specify a packed decimal string, but note that L is the number of digits, not bytes, in the string. Every byte of a packed decimal string is divided into two 4-bit fields (nibbles), each of which must contain decimal digits, except the low nibble of the last byte, which must contain a sign. The representation for the digits and sign is:

digit or sign	decimal	hex
0	0	0
1	1	1
2	2	2
3	3	3
4	4	4
5	5	5
6	6	6
7	7	7
8	8	8
9	9	9
+	10, 12, 14 or 15	A, C, E, or F
–	11 or 13	B, or D

Despite the options, the preferred sign representation is 12 for + and 13 for –. The length L is the number of digits in the packed decimal string (not counting the sign) and must be in the range 0 through 31. If the number of digits is odd, the digits and the sign fit into ([L/2]+1) bytes; when the number of digits is even, an extra "0" digit must appear in the high nibble (bits 7:4) of the first byte. Again, the length in bytes of the string is L/2 + 1. The value of a 0-length packed decimal string is identically 0; it contains only the sign byte which also includes the extra "0" digit.

The address A of the string specifies the byte of the string containing the most significant digit in its high nibble. Digits of decreasing significance are assigned to increasing byte addresses and from high nibble to low nibble within a byte. In the following example, +123 (length 3) is represented in packed decimal format as:

7 4	3 0	
1	2	: A
3	12	: A+1

and −**12** (length 2) is represented as:

7 4	3 0	
0	1	: A
2	13	: A+1

VARIABLE LENGTH BIT FIELD DATA TYPE

The variable length bit field is a data type used to store small integers packed together in a larger data structure. This saves memory when many small integers are part of a larger structure. A specific case of the variable bit field is that of one bit. This form is used to store and access individual flags efficiently.

Variable Length Bit Field

A variable bit field is 0 to 32 contiguous bits located arbitrarily with respect to byte boundaries, and specified by three attributes:

- Base address A—the address of a particular byte in memory chosen as a reference point for locating the bit field F.
- Bit position P—the signed longword specifying the bit displacement of the least significant bit of the field with respect to bit zero of the byte at address A.
- Size S—the byte integer length of field F expressed as a number of bits. S must be between 0 and 32 bits inclusive.

Figure 4-5a illustrates the variable length bit field where the field is the shaded area.

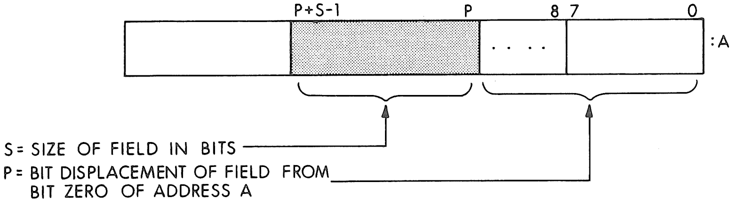


Figure 4-5a Variable Length Bit Field

The position P (in bits) can be either a positive or negative displacment within the range -2^{31} through $2^{31}-1$. It can be viewed as a signed 29-bit byte offset and a 3-bit bit-within-byte field.

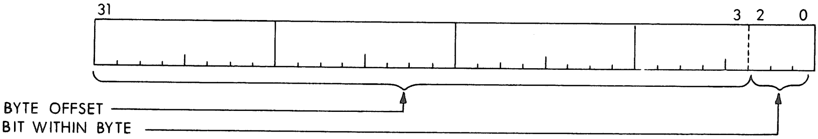


Figure 4-5b Bit Position P

The sign-extended 29-bit byte offset is added to the address A and the resulting address specifies the byte in which the field begins. The 3-bit bit-within-byte field encodes the starting position (0 through 7) of the field within that byte. VAX instructions provide direct support for the interpretation of a field as a signed or unsigned integer. When interpreted as a signed integer, it is the 2's complement, with bits increasing in significance from 0 through S−2, where bit S−1 is designated the sign bit. When interpreted as an unsigned integer, bits increase in significance from 0 through S−1.

If the field is contained in a register, and the size is not zero, the position operand must have a value in the range 0 through 31 or a reserved operand fault occurs.

If size plus position are greater than 32, then the operand is located in the concatenation of register [n+1] and by register [n] (i.e., R[n+1]'R[n]). Therefore, the most significant bit of the specified field lies in R[n+1] and the least significant bit of the specified field is located in R[n].

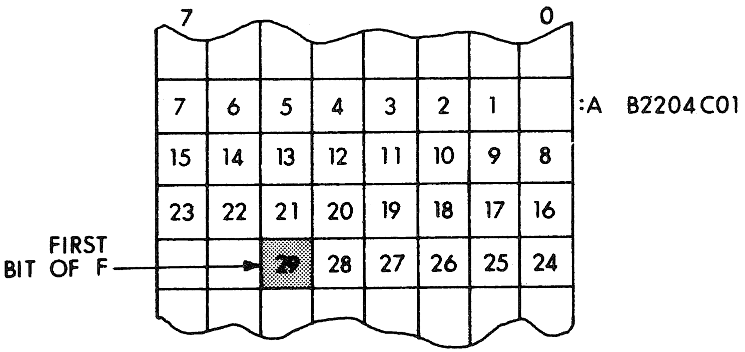
A variable bit field may be contained in zero to five bytes. From a memory management point of view only the minimum number of bytes necessary to contain the field is actually referenced.

The following example illustrates the variable length bit field F with a positive displacement from the byte address A.

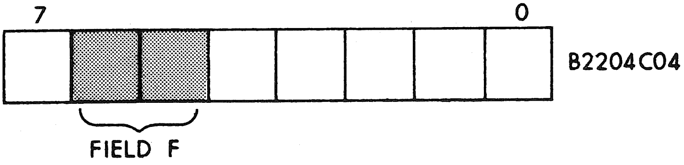
The variable length bit field attributes are specified as follows:

Base Address A = B2204C01
Position P = 29
Size S = 2

Therefore, the starting position of the field is bit 29 (i.e., the first bit of F is the 29th bit after bit 0 of A).



The starting bit position of field F has been located. To determine its length, apply the size attribute.



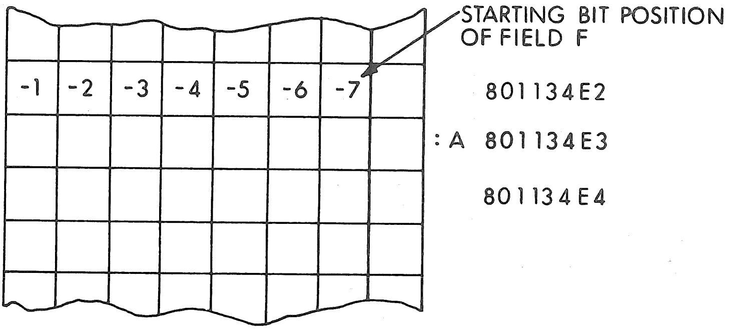
The next example illustrates the variable length bit field F with a negative displacement from the byte address A.

Example:

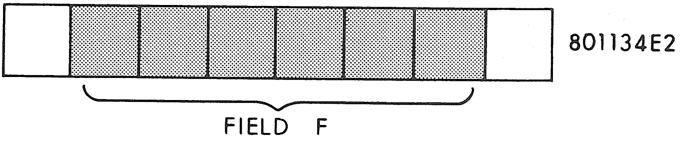
The variable length bit field attributes are specified as follows:

Base Address A = 801134E3
Position P = −7
Size S = 6

Therefore, the starting position of field F is the 7th bit preceding the zero bit of address 801134E3.



The starting bit position of field F has been located. To determine its length, apply the size attribute as in the previous example (counting from lower to higher addresses).



QUEUE DATA TYPES

A queue is a circular, doubly linked list whose entries are specified by their addresses. Each queue entry is linked to the next via a pair of longwords. The first is the forward link--it specifies the location of the succeeding entry; the second is the backward link--it specifies the location of the preceding entry. VAX supports two distinct types of links: 1) absolute and 2) self-relative. An absolute link contains the absolute address of the entry that it points to, while a self-relative link contains a displacement from the present queue entry. A queue requires a queue header which is identical to a pair of queue linkage longwords. The forward link of the header is the address of the entry termed the head of the queue. The backward link of the header is the address of the entry termed the tail of the queue. Logically, the forward link of the tail points to the header.

Self-relative queues are intended for use in situations where they are addressed by two separate processes, each of which may view the queues as residing in two separate locations in their respective virtual address spaces. The instructions which operate on self-relative queues are interlocked: as long as only interlocked instructions are used on the queue, the processes may be in separate processors, each directly addressing the queue.

Absolute queues are somewhat simpler in structure than self-relative queues in that their pointers are virtual addresses. Also, the instructions which operate on these queues are not interlocked. Hence, operations on absolute queues are, in general, somewhat faster. However, absolute queues cannot be used when more than one processor can access them. Also, they can be shared by two processes in the same processor only when both processes address the queue in the same section of their virtual address space. Figure 4-6a illustrates the format of the self-relative queue and Figure 4-6b illustrates the format of the absolute queue.

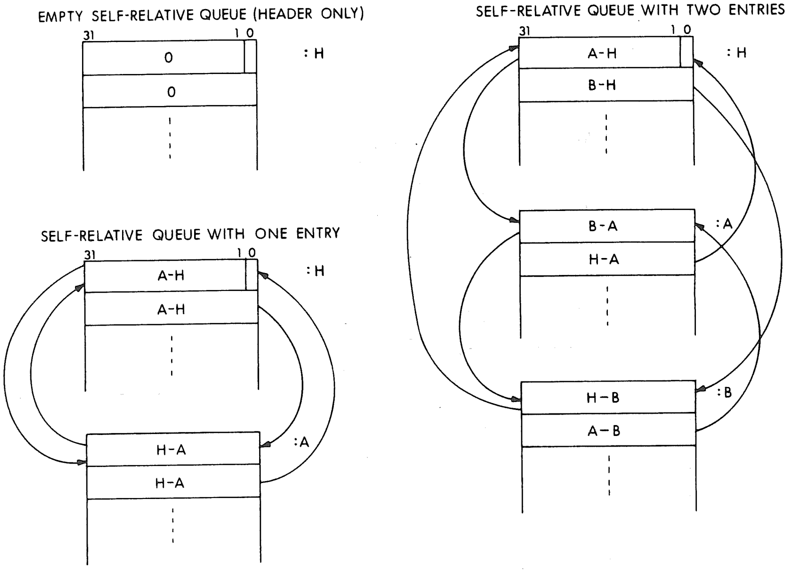
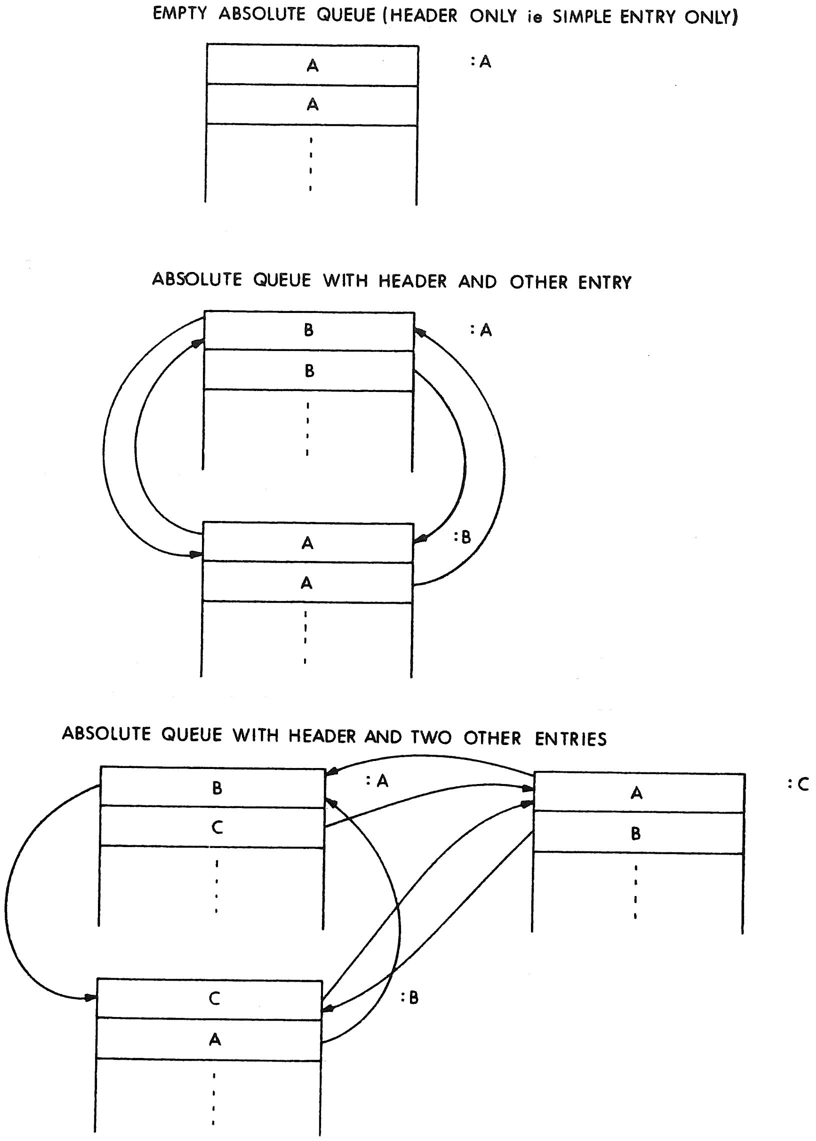

EMPTY SELF-RELATIVE QUEUE (HEADER ONLY)
31
1 0
0
0
: H
SELF-RELATIVE QUEUE WITH ONE ENTRY
31
1 0
A-H
A-H
: H
H-A
H-A
:A
SELF-RELATIVE QUEUE WITH TWO ENTRIES
31
1 0
A-H
B-H
: H
B-A
H-A
: A
H-B
A-B
: B

Figure 4-6a Self-Relative Queues



EMPTY ABSOLUTE QUEUE (HEADER ONLY ie SIMPLE ENTRY ONLY)

ABSOLUTE QUEUE WITH HEADER AND OTHER ENTRY

ABSOLUTE QUEUE WITH HEADER AND TWO OTHER ENTRIES

Figure 4-6b Absolute Queues

DATA IN REGISTERS

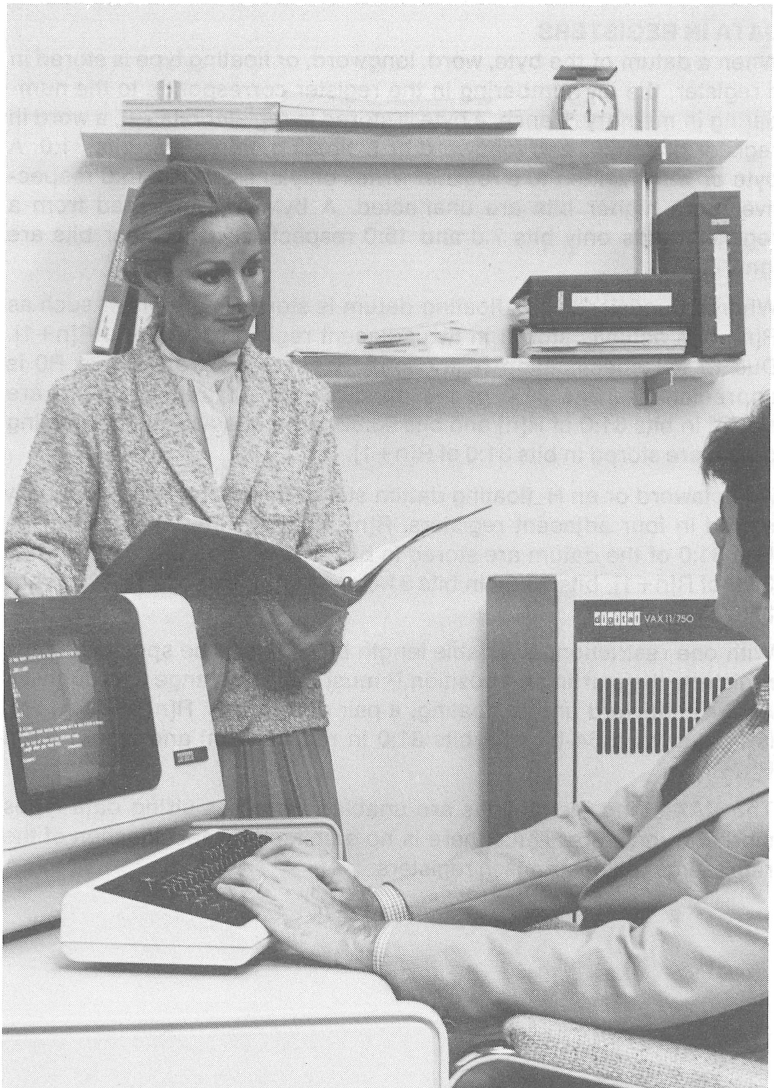
When a datum of the byte, word, longword, or floating type is stored in a register, the bit numbering in the register corresponds to the numbering in memory. Hence, a byte is stored in register bits 7:0, a word in register bits 15:0, and longword or F_floating, in register bits 31:0. A byte or word written to a register writes only bits 7:0 and 15:0 respectively; the higher bits are unaffected. A byte or word read from a register reads only bits 7:0 and 15:0 respectively; the other bits are ignored.

When a quadword or D_floating datum is stored in a register such as R[n], it is actually stored in two adjacent registers, R[n] and R[n+1]. Due to PC specification restrictions, wraparound from PC to R0 is unpredictable. Bits 31:0 of the quadword or D_floating datum are stored in bits 31:0 of R[n] and bits 63:32 of the quadword or D_floating datum are stored in bits 31:0 of R[n+1].

An octaword or an H_floating datum stored in register R[n], is actually stored in four adjacent registers, R[n], R[n+1], R[n+2], and R[n+3]. Bits 31:0 of the datum are stored in bits 31:0 of R[n], bits 63:32 in bits 31:0 of R[n+1], bits 95:64 in bits 31:0 of R[n+2], and bits 127:96 in bits 31:0 of R[n+3.

With one restriction, a variable length bit field may be specified in the registers: the starting bit position P must be in the range 0 through 31. As for quadword and D_floating, a pair of registers, R[n] and R[n+1], is treated as a 64-bit with bits 31:0 in register R[n] and bits 63:32 in R[n+1].

The VAX string instructions are unable to process string data types stored in registers. Thus, there is no architectural specification of the representation of strings in registers.



VAX 11/750

CHAPTER 5

INSTRUCTION FORMATS AND ADDRESSING MODES

INTRODUCTION

The addressing modes together with 16 general-purpose registers provide a convenient method of manipulating data by specifying how the selected registers are used to access, manipulate, and store data and instructions in memory.

GENERAL REGISTERS

VAX general-purpose registers can be used with an instruction in any of the following ways:

- As accumulators. The data to be processed are contained in the register.
- As pointers. The address of the operand, rather than the operand itself, is the content of the register. This form is often referred to as a base register because it frequently contains the base address of a data structure.
- As pointers which automatically step through memory locations. Stepping forward automatically through consecutive locations is known as autoincrement addressing; automatically stepping backwards is known as autodecrement addressing. These modes are particularly useful for processing tabular data and manipulating stacks.
- As index registers. In index mode, an offset is generated and then added to the base operand address to yield the indexed location.

One of the general-purpose registers is designated a **stack pointer** and provides temporary storage for data which are frequently accessed. In the VAX, any register can be used as a stack pointer under program control; however, certain instructions associated with subroutine linkage and interrupt service automatically use R14 as a "hardware stack pointer." For this season, R14 is called the SP.

Stack pointer addresses **decrease** as items are added to the stack. This is conveniently done by decrementing the address and "pushing" data on the stack. On the other hand, stack pointer addresses increase as items are removed from the stack. This is conveniently done by incrementing the address and "popping" data from the stack. Consequently, the stack pointer always points to the lowest addressed end of the stack. The hardware stack is used during exception or interrupt handling to store breakpoint information, allowing the processor to return to the main program.

R15 is used by the processor as a program counter (PC) which points to the next instruction in the program to be executed. Whenever an instruction is fetched from memory, the program counter is automatically incremented by the number of bytes in the instruction.

INSTRUCTION FORMAT

The VAX instruction set has a variable length instruction format which may be as short as one byte and as long as needed depending on the type of instruction. Shown in Figure 5-1 is a schematic of the general format. Each instruction consists of an opcode followed by zero to six operand specifiers whose number and type depend on the opcode. All operand specifiers are, themselves, of the same format—i.e., an address mode plus additional information. This additional information contains up to two register designators and addresses, data, or displacements. The operand usage is determined implicitly from the opcode, and is termed the **operand type**. It includes both the access type and the data type. Figure 5-2 shows several examples of VAX instruction formats.

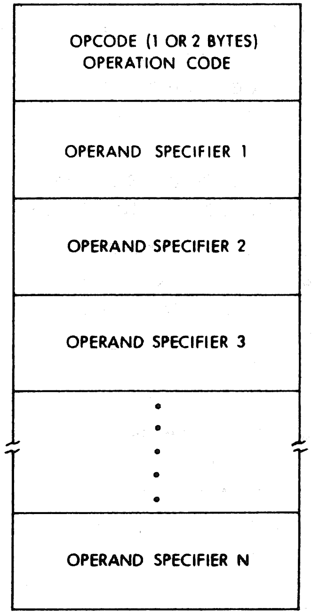


Figure 5-1 General VAX Instruction Format

A. MOVE LONG INSTRUCTION

```
MOVL  6(R1), R5    ; SIX IS ADDED TO R1, THE RESULT USED AS AN
                   ; ADDRESS AND THE CONTENTS OF THAT ADDRESS
                   ; IS MOVED TO R5
```

BYTE		
1	MOVL	OPCODE
2	(R1)	OPERAND SPECIFIER 1
3	6	
4	R5	OPERAND SPECIFIER 2

B. MOVE WORD INSTRUCTION

```
MOVW #↑X3456, -(SP)    ; THE NUMBER 3456 IS PUSHED ON THE
                       ; STACK
```

BYTE		
1	MOVW	OPCODE
2	(PC) +	OPERAND SPECIFIER 1
3	56	IMMEDIATE DATA (56 STORED IN BYTE 3)
4	34	(34 STORED IN BYTE 4)
5	-(SP)	OPERAND SPECIFIER 2

C. ADD LONG INSTRUCTION (3 OPERAND)

```
ADDL3  (SP)+, R4, R5    ; NUMBER ON THE STACK IS
                        ; ADDED TO THE CONTENTS OF
                        ; R4 AND RESULT IS STORED
                        ; IN R5
```

BYTE		
1	ADDL 3	OPCODE
2	(SP) +	OPERAND SPECIFIER 1
3	R4	OPERAND SPECIFIER 2
4	R5	OPERAND SPECIFIER 3

Figure 5-2   Examples of Instruction Format

Assembler Radix Notation

The radix of the assembler is decimal. To express a hexadecimal number in assembler notation, it is required to precede the number by ↑X. For example, the assembler interprets the 3456 in "MOVW #3456, −(SP)" as a decimal number. If it is to be interpreted as a hexadecimal number, it would be written

MOVW #↑X 3456, −(SP)

Conversion algorithms, hexadecimal arithmetic tables, and a decimal/hexadecimal conversion chart are provided in Appendix A.

Operating Code (Opcode)
Each VAX instruction contains an opcode which specifies the desired operation to be performed. The opcode may be one or two bytes long, depending on the contents of the byte at address A. Two bytes will be used under the condition that the value of that byte is FC(hex) through FF(hex).

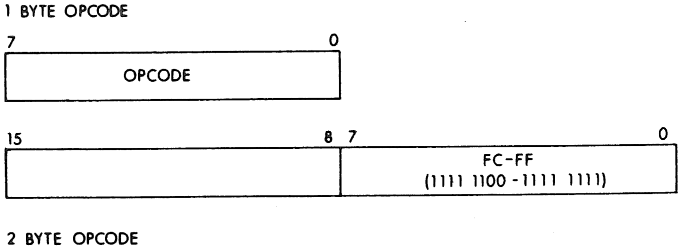


Figure 5-3 Opcode Format

Operand Types
The operand types specify how the operand associated with an instruction is used. Information derived from the opcode includes the data type of each operand and how the operand is accessed. The data types include:

- Byte—8 bits
- Word—16 bits
- Longword and F_floating—32 bits (equivalent for addressing mode considerations)
- Quadword, D_floating, and G_floating—64 bits (similarly equivalent)
- Octaword and H_floating—128 bits (similarly equivalent)

An operand may be accessed in one of six ways. They are:

1. Read—The specified operand is read-only.
2. Write—The specified operand is write-only.
3. Modify—The specified operand is read, may or may not be modified, and is written.
4. Address—Address calculation occurs until the actual address of the operand is obtained. In this mode, the data type indicates the operand size to be used in the address calculation. The specified operand is not accessed directly although the instruction may subsequently use the address to access that operand.

5. Variable bit field base address—If just R[n] is specified, the field is in general register n or in R[n + 1]'R[n] (i.e., R[n + 1] concatenated with R[n]). Otherwise, address calculation occurs until the actual address of the operand is obtained. This address specifies the base to which the field position (offset) is applied.
6. Branch—No operand is accessed. The operand specifier *itself* is a branch displacement. In this specifier, the data type indicates the size of the branch displacement.

Operand Specifier

An operand specifier gives the information needed to locate the operand. Each general mode addressing description includes the definition of the operand address and the specified operand. For operand specifiers of address access type, the operand address is the actual instruction operand; for other access types, the specified operand is the instruction operand. The branch mode addressing description includes the definition of the branch address.

ADDRESSING MODE

Broadly speaking, VAX addressing can be divided into two categories, general mode addressing and branch addressing. The sections that follow describe the various modes under both general types.

Table 5-1, below, is a quick summary of the general register and program counter addressing modes. It shows: the mode specifier for each addressing mode in hexadecimal and decimal notation; the assembler notation; the access types which may be used with the various modes; the effect on the program and stack pointer; and which modes may be indexed. For example, in literal mode, only a read access may occur. Any other type of access results in a fault. (The program counter and stack pointer are not referenced in this mode and are logically impossible. If indexing is attempted in this mode, a reserved addressing mode fault will occur.)

Following the chart is a more detailed explanation of each of these modes, and following those, detailed explanations of branch addressing.

Table 5-1 Summary of Addressing Modes

GENERAL REGISTER ADDRESSING

Hex	Dec	Name	Assembler	r	m	w	a	v	PC	SP	Indexable?
0-3	0-3	literal	S^#literal	y	f	f	f	f	—	—	f
4	4	indexed	i [Rx]	y	y	y	y	y	f	y	f
5	5	register	Rn	y	y	y	f	y	u	uq	f
6	6	register deferred	(Rn)	y	y	y	y	y	u	y	y
7	7	autodecrement	—(Rn)	y	y	y	y	y	u	y	ux
8	8	autoincrement	(Rn)+	y	y	y	y	y	p	y	ux
9	9	autoincrement deferred	@ (R)+	y	y	y	y	y	p	y	ux
A	10	byte displacement	B^D (Rn)	y	y	y	y	y	p	y	y
B	11	byte displacement deferred	@B^D (Rn)	y	y	y	y	y	p	y	y
C	12	word displacement	W^D (Rn)	y	y	y	y	y	p	y	y
D	13	word displacement deferred	@W^D (Rn)	y	y	y	y	y	p	y	y
E	14	longword displacement	L^D (Rn)	y	y	y	y	y	p	y	y
F	15	longword displacement deferred	@L^D (Rn)	y	y	y	y	y	p	y	y

PROGRAM COUNTER ADDRESSING

Hex	Dec	Name	Assembler	r	m	w	a	v	PC	SP	Indexable?
8	8	immediate	I^#constant	y	u	u	y	y	—	—	y
9	9	absolute	@#address	y	y	y	y	y	—	—	y
A	10	byte relative	B^address	y	y	y	y	y	—	—	y
B	11	byte relative deferred	@B^address	y	y	y	y	y	—	—	y
C	12	word relative	W^address	y	y	y	y	y	—	—	y
D	13	word relative deferred	@W^address	y	y	y	y	y	—	—	y
E	14	longword relative	L^address	y	y	y	y	y	—	—	y
F	15	longword relative deferred	@L^address	y	y	y	y	y	—	—	y

D — displacement
i — any indexable addressing mode
- — logically impossible
f — reserved addressing mode fault
p — Program Counter addressing
u — unpredictable
uq — unpredictable for quadword, octaword, D_floating, G_floating, and H_floating (and field, if position + size greater than 32)
uo — unpredictable for octaword, and H_floating format.
ux — unpredictable for index register same as base register
y — yes, always valid addressing mode
r — read access
m — modify access
w — write access
a — address access
v — field access

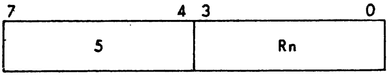
GENERAL MODE ADDRESSING

Register Mode

Assembler Syntax: Rn

Mode Specifier: 5

Operand Specifier Format:



The operand is the content of register n (or R[n+1] concatenated with Rn for quadword, D_floating, and certain field operations):

Operand = Rn if one register, or
R[n+1]'R[n] if two registers, or
R[n+3]'R[n+2]'R[n+1]'R[n] if four registers

Description: With register mode, any of the general registers may be used as simple accumulators and the operand is contained in the selected register. Since they are hardware registers within the processor, they provide speed advantages when used for operating on frequently-accessed variables.

Special Comments: This mode can be used with operand specifiers using read, write or modify access but cannot be used with the address access type; otherwise, an illegal addressing mode fault results. The program counter (PC) cannot be used in this mode. If the PC is read, the value is unpredictable; if the PC is written, the next instruction executed or the next operand specified is unpredictable. Similarly, if PC is used in register mode for a write-access operand which takes two adjacent registers, the contents of R0 are unpredictable.

The stack pointer, (SP), cannot be used in this mode for an operand which takes two adjacent registers since that would imply a direct reference to the PC and the results are unpredictable.

EXAMPLE: REGISTER MODE, MOVE WORD INSTRUCTION

Instruction Format: MOVW R1, R2

Instruction moves a 16-bit word of data from R1 to R2.

BEFORE INSTRUCTION EXECUTION

R1							
C	0	A	0	3	4	1	2

R2							
0	0	0	0	0	0	0	0

AFTER INSTRUCTION EXECUTION

C	0	A	0	3	4	1	2

0	0	0	0	3	4	1	2

MACHINE CODE: ASSUME STARTING LOCATION 00003000

```
00003000    B0    OPCODE FOR MOVE WORD INSTRUCTION
00003001    51    OPERAND SPECIFIER,SOURCE; REGISTER MODE 1
00003002    52    OPERAND SPECIFIER, DESTINATION; REGISTER MODE 2
```

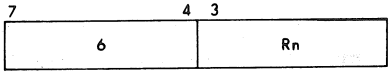
This example shows a Move Word instruction using register mode. The content of R1 is the operand and the Move Word instruction causes the least significant half of R1 to be transferred to the least significant half of register R2. The upper half of R2 is unaffected.

Register Deferred Mode

Assembler Syntax: (Rn)

Mode Specifier: 6

Operand Specifier Format:



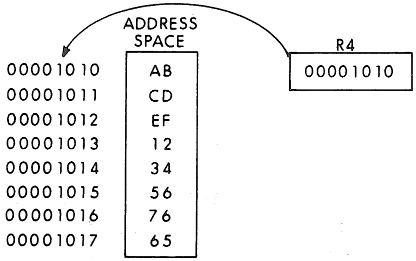
Description: The register deferred mode provides one level of indirect addressing over register mode; that is, the general register contains the address of the operand rather than the operand itself. The deferred modes are useful when dealing with an operand whose address is calculated.

Special Comments: The PC cannot be used in register deferred mode addressing as the results will be unpredictable.

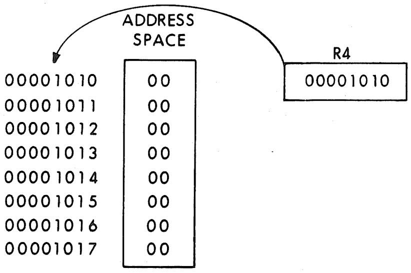
EXAMPLE: REGISTER DEFERRED MODE, CLEAR QUAD INSTRUCTION

Instruction Format: CLRQ (R4)

BEFORE INSTRUCTION EXECUTION



AFTER INSTRUCTION EXECUTION



MACHINE CODE: ASSUME STARTING LOCATION 00003000

00003000	7C	OPCODE FOR CLEAR QUAD INSTRUCTION
00003001	64	OPERAND SPECIFIER FOR REGISTER DEFERRED MODE, R4

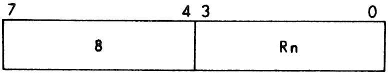
This example shows a Clear Quad instruction using Register Deferred Mode. R4 contains the address of the operand and the instruction specifies that the byte at this address plus the following seven bytes are to be cleared.

Autoincrement Mode

Assembler Syntax: (Rn)+

Mode Specifier: 8

Operand Specifier Format:



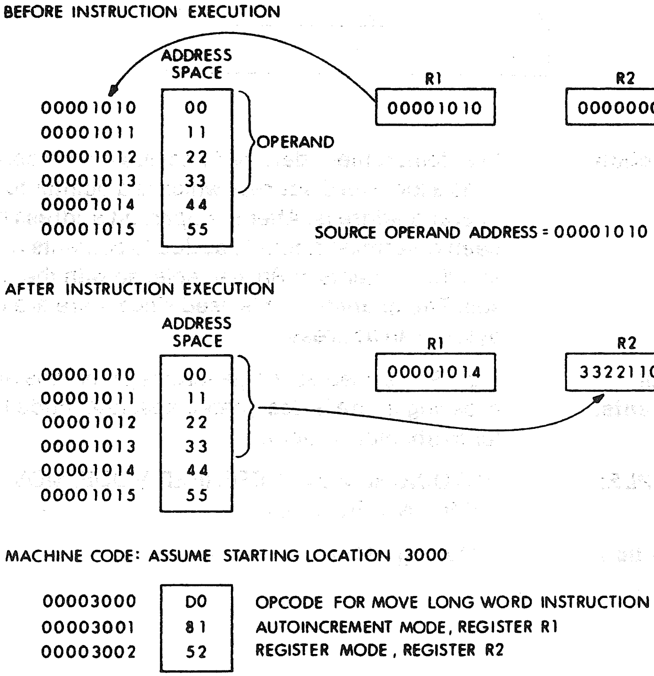
Description: In autoincrement mode addressing, Rn contains the address of the operand. After the operand address is determined, the size of the operand (which is determined by the instruction) in bytes (1 for byte, 2 for word, 4 for longword or F_floating, 8 for quadword, D_floating, or G_floating, and 16 for octaword or H_floating) is added to the contents of Rn and the contents of Rn are replaced by the result. This mode provides for automatic stepping of a pointer through sequential elements of a table of operands. Contents of registers are incremented to address the next sequential location. The autoincrement mode is especially useful for array processing and stacks. It will access an element of a table and then step the pointer to address the next operand in the table. Although most useful for table handling, this mode is general and may be used for variety of purposes.

Special Comments: If the PC is used as the general register, this addressing mode is designated immediate mode and has special syntax (refer to immediate mode).

EXAMPLE: AUTOINCREMENT MODE, MOVE LONG INSTRUCTION

Instruction Format: MOVL (R1)+, R2

This instruction will move a longword of data (32 bits) to R2.


BEFORE INSTRUCTION EXECUTION
ADDRESS SPACE
00001010
00001011
00001012
00001013
00001014
00001015
00
11
22
33
44
55
OPERAND
R1
00001010
R2
00000000
SOURCE OPERAND ADDRESS = 00001010
AFTER INSTRUCTION EXECUTION
ADDRESS SPACE
00001010
00001011
00001012
00001013
00001014
00001015
00
11
22
33
44
55
R1
00001014
R2
33221100
MACHINE CODE: ASSUME STARTING LOCATION 3000
00003000
00003001
00003002
D0
81
52
OPCODE FOR MOVE LONG WORD INSTRUCTION
AUTOINCREMENT MODE, REGISTER R1
REGISTER MODE, REGISTER R2

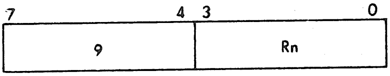
This example shows a Move Long instruction using autoincrement mode. The content of R1 is the effective address of the source operand. Since the operand is a 32-bit longword, four bytes are transferred to R2. R1 is then incremented by four since the instruction specifies a longword data type.

Autoincrement Deferred Mode

Assembler Syntax: @(Rn)+

Mode Specifier: 9

Operand Specifier Format:



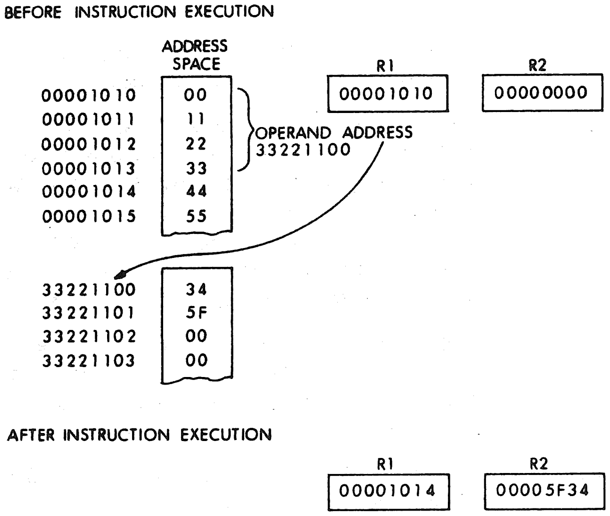
Description: In autoincrement deferred addressing, Rn contains a longword address which is a pointer to the operand address. After the operand address has been determined, four is added to contents of Rn and the contents of Rn are replaced with the result. The quantity four is used since there are four bytes in an address.

Special Comments: If the PC is used as the general register, this addressing mode is designated absolute mode (refer to absolute mode).

EXAMPLE: AUTOINCREMENT DEFERRED MODE, MOVE WORD INSTRUCTION

Instruction Format: MOVW @(R1)+, R2

Operand Specifier Format:



MACHINE CODE: ASSUME STARTING LOCATION 00003000

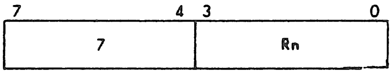
```
00003000   B0   OPCODE FOR MOVE WORD INSTRUCTION
00003001   91   AUTOINCREMENT DEFERRED MODE, REGISTER R1
00003002   52   REGISTER MODE, REGISTER R2
```

This example shows a Move Word instruction using autoincrement deferred mode. The content of R1 is a pointer to the operand address. Since a word length instruction is specified, the byte at the effective address and the byte at the effective address plus one are loaded into the low-order half of register R2; the upper half of R2 is unaltered. R1 is then incremented by four since it points to a 32-bit address.

Autodecrement Mode

Assembler Syntax: −(Rn)

Mode Specifier: 7

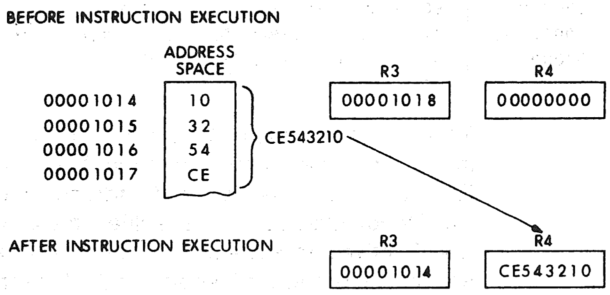


The contents of Rn are decremented and then used as the address of the operand.

Description: With autodecrement mode, the size of the operand in bytes (1 for byte, 2 for word, 4 for longword or F_floating, 8 for quadword, G_floating, or D_floating, and 16 for octaword or H_floating) is subtracted from the content of Rn and the content of Rn is replaced by the result. The updated content of Rn is the address of the operand.

Special Comments: The PC may not be used in autodecrement mode. If it is, the address of the operand is unpredictable and the next instruction executed or the next operand specifier is upredictable.

EXAMPLE: AUTODECREMENT MODE, MOVE LONG INSTRUCTION MOVL −(R3), R4



MACHINE CODE: ASSUME STARTING LOCATION 00003000

00003000	D0	OPCODE FOR MOVE LONG INSTRUCTION
00003001	73	AUTODECREMENT MODE, REGISTER R3
00003002	54	REGISTER MODE, REGISTER R4

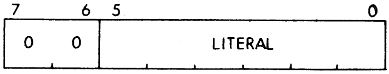
This example shows a Move Long instruction using autodecrement mode. The contents of R3 are decremented according to the data type specified in the opcode (four in this example because a longword is used). The updated contents of R3 are then used as the address of the operand. The instruction causes the operand to be fetched and loaded into R4.

Literal Mode

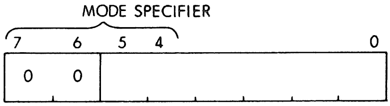
Assembler Syntax: S↑# literal

Mode Specifier: 0, 1, 2 or 3
(depending on literal value specified)

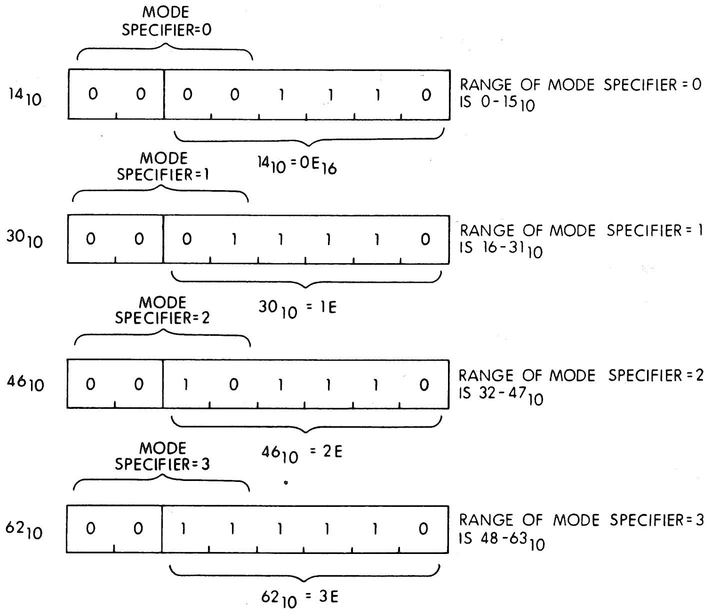
Operand Specifier Format:



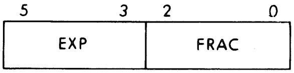
Description: Literal mode addressing provides an efficient means of specifying integer constants in the range from 0 to 63 (decimal). This is called short literal. Literal values above 63 can be obtained by immediate mode (autoincrement mode using the PC) although immediate mode is longer. For predefined values, the assembler will choose between short literal and immediate modes. For short literal operands, the format is:



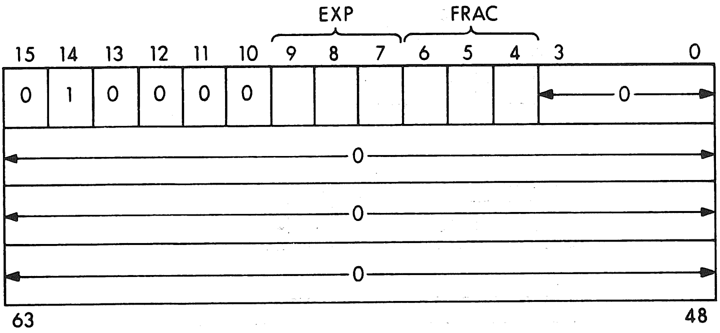
Bits 7 and 6, however, are always set to zero. The following examples show some short literals; the literals are 14, 30, 46, and 62.



Floating point literals as well as short literals can be expressed. The floating point literals are listed in Table 5-2. For operands of the short floating type, the 6-bit literal field in the operand specifier is composed of two 3-bit fields where EXP designates exponent and FRAC designates the fraction.



The 3-bit EXP field and 3-bit FRAC field are used to form an F_floating or D_floating operand as follows, where bits 63:32 are not present in an F_floating operand:



NOTE
G_floating and H_floating operands can be formed in analogous ways using the EXP and FRAC fields.

Bits 3 through 5 of the EXP field are stored in bits 7 through 9, respectively, of the floating operand. Bits 0 through 2 of the FRAC field are stored in bits 4 through 6, respectively, in the floating operand. The actual decimal values which can be stored are given in Table 5-2.

The EXP field is expressed in "excess 128" notation. In this notation, an offset of 128 is actually added to the exponent. For example, an exponent of zero is represented as 128 or 10000000 (binary), while an exponent of three is represented as 131 or 10000011 (binary).

Assume you want to express the floating point literal of 12. Table 5-2 shows the decimal literal of 12 to be represented by a fraction of 4 and an exponent of 4.

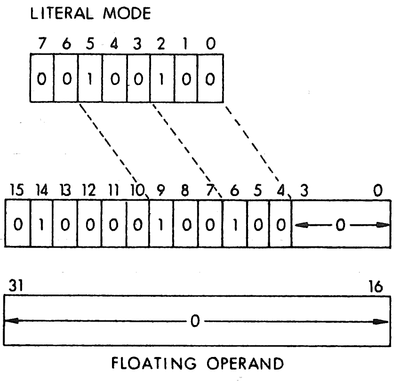


Table 5-2 Floating Literals

Exponent	FRACTION							
	0	1	2	3	4	5	6	7
0	½	9/16	5/8	11/16	3/4	13/16	7/8	15/16
1	1	1 1/8	1¼	1 3/8	1½	1 5/8	1¾	1 7/8
2	2	2¼	2½	2¾	3	3¼	3½	3¾
3	4	4½	5	5½	6	6½	7	7½
4	8	9	10	11	12	13	14	15
5	16	18	20	22	24	26	28	30
6	32	26	40	44	48	52	56	60
7	64	72	80	88	96	104	112	120

EXAMPLE: **LITERAL MODE, MOVE LONG INSTRUCTION**
MOVL S↑#9, R4

BEFORE INSTRUCTION EXECUTION

R4

00000000

AFTER INSTRUCTION EXECUTION

R4

00000009

MACHINE CODE: ASSUME STARTING LOCATION 00003000

```
00003000   D0   OPCODE FOR MOVE LONG INSTRUCTION
00003001   09   LITERAL 9
00003002   54   REGISTER MODE, REGISTER R4
```

This example shows a Move Long instruction using literal mode. The literal 9 is transferred to register R4 as a result of the instruction.

Displacement Mode

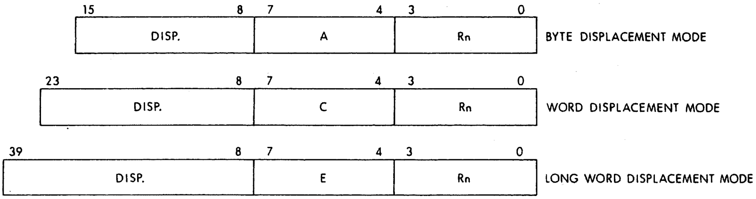
Assembler Syntax:

D(Rn)—general displacement syntax
B↑D(Rn)—forces byte displacement
W↑D(Rn)—forces word displacement
L↑D(Rn)—forces longword displacement

Mode Specifier:

A—(byte displacement)
C—(word displacement)
E—(longword displacement)

Operand Specifier Format:



Description:

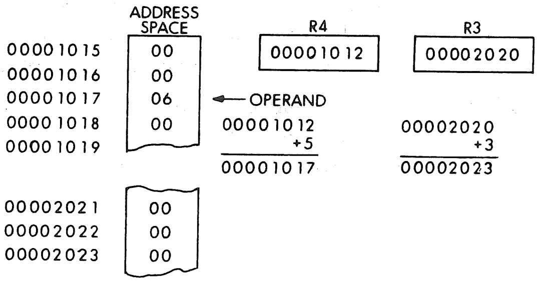
In displacement mode addressing, the displacement (after being sign-extended to 32 bits if it is a byte or word) is added to the content of register Rn and the result is the operand address. This mode is the equivalent of index mode in the PDP-11 series.

The VAX architecture provides for an 8-bit, 16-bit, or 32-bit offset. Since most program references occur within small discrete portions of the address space, a 32-bit offset is not always necessary and the 8- and 16-bit offsets will result in substantial economies of space (that is, fewer bits are required).

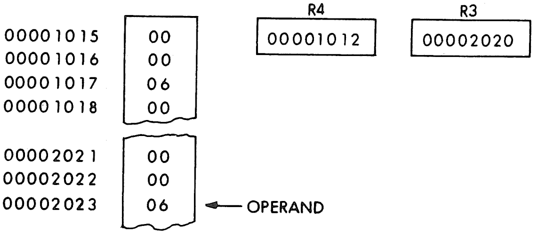
If the PC is used as the general register, this mode is called relative mode (refer to relative mode).

EXAMPLE: DISPLACEMENT MODE, MOVE BYTE INSTRUCTION
MOVB B↑5(R4), B↑3(R3)

BEFORE INSTRUCTION EXECUTION



AFTER INSTRUCTION EXECUTION



MACHINE CODE: ASSUME STARTING LOCATION 00003000

00003000	90	OPCODE FOR MOVE BYTE INSTRUCTION
00003001	A4	SIGNED BYTE DISPLACEMENT, REGISTER R4
00003002	05	SPECIFIER EXTENSION (DISPLACEMENT OF 5)
00003003	A3	SIGNED BYTE DISPLACEMENT, REGISTER R3
00003004	03	SPECIFIER EXTENSION (DISPLACEMENT OF 3)

This example shows a Move Byte instruction using displacement mode. A displacement of 5 is added to the content of R4 to form the address of the byte operand. The operand is moved to the address formed by adding the displacement of 3 to the contents of R3.

Displacement Deferred Mode

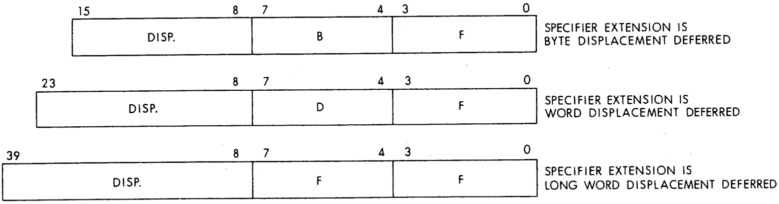
Assembler Syntax:

@D(Rn)
@B↑D(Rn) byte displacement deferred
@W↑D(Rn) word displacement deferred
@L↑D(Rn) longword displacement deferred

Mode Specifier:

B—(byte displacement)
D—(word displacement)
F—(longword displacement)

Operand Specifier Format:



Description:

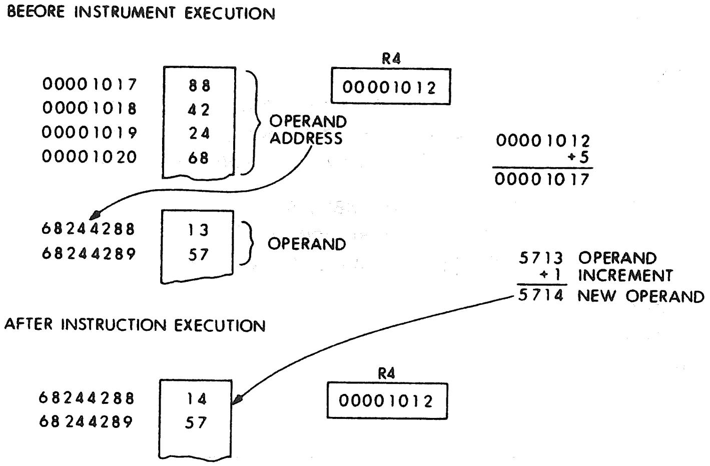
In displacement deferred mode addressing, the displacement (after being sign-extended to 32 bits if it is a byte or word) is added to the content of the selected general Rn and the result is a longword address of the operand address.

If the PC is used as the general register, this mode is called relative deferred mode (refer to relative deferred mode).

EXAMPLE:

DISPLACEMENT DEFERRED MODE, INCREMENT WORD INSTRUCTION
INCW @B↑5(R4)

This example shows an Increment Word instruction using displacement deferred mode. The quantity 5 is added to the contents of R4 to produce the longword address of the address of the operand. The operand of 5713 is incremented to 5714.



MACHINE CODE: ASSUME STARTING LOCATION 00003000

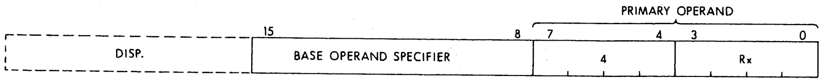
00003000	B6	OPCODE FOR INCREMENT WORD INSTRUCTION
00003001	B4	SIGNED BYTE DISPLACEMENT, REGISTER R4
00003002	05	SPECIFIER EXTENSION REGISTER R4 PLUS SIGN

Index Mode

Assembler Syntax: i[Rx]

Mode Specifier: 4

Operand Specifier Format:



Description:

The operand specifier consists of at least two bytes—a primary operand specifier and a base operand specifier. The primary operand specifier contained in bits 0 through 7 includes the index register (Rx) and a mode specifier of 4. The address of the primary operand is determined by first multiplying the contents of index register Rx by the size of the primary operand in bytes (1 for byte, 2 for word, 4 for longword or F_floating, 8 for quadword, D_floating, or G_floating, and 16 for octaword or H_floating). This value is then added to the address specified by the base operand specifier (bits 15:8), and the result is taken as the operand address.

The chief advantage of index mode addressing is to provide very general and efficient accessing of arrays. VAX architecture provides for context indexing where the number in the index register is shifted left by the context of the data type specified (none for byte, once for word, twice for longword, three times for quadword and four times for octaword). This allows loop control variables to be used in the address calculation without first shifting them the appropriate number of times, thus minimizing the number of instructions required. This feature is used to advantage in the FORTRAN IV-PLUS compiler.

Specifying register, literal, or index mode for the base operand specifier will result in an illegal addressing mode fault. If the use of some particular specifier is illegal (causes a fault or unpredictable behavior), then that specifier is also illegal as a base operand specifier in index mode under the same conditions.

Special Comments:

The following restrictions are placed on index register Rx:

1. The PC cannot be used as an index register. If it is, a reserved addressing mode fault occurs.
2. If the base operand specifier is for an addressing mode which results in register modification (autoincrement, autoincrement de-

ferred, or autodecrement), the same register cannot be the index register. If it is, the primary operand address is unpredictable.

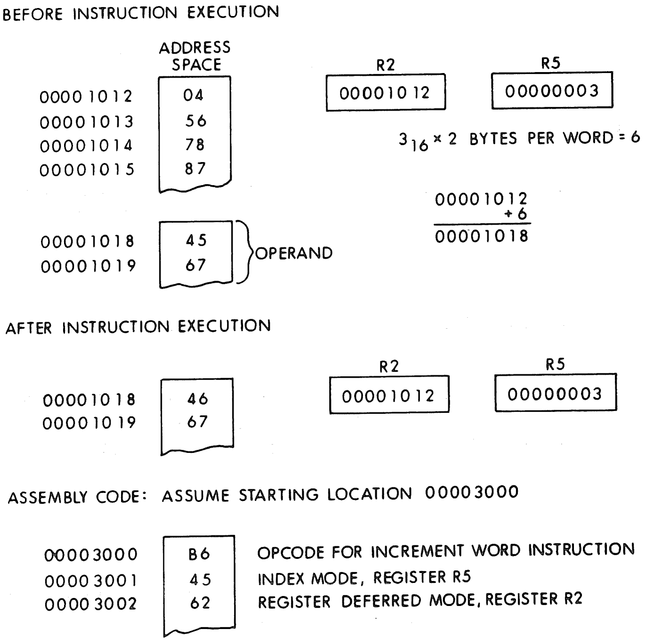
Table 5-3 lists the various forms of index mode addressing available. The names of the addressing modes resulting from index mode addressing are formed by adding **indexed** to the addressing mode of the base operand specifier. The general register is designated Rn and the indexed register is Rx.

Table 5-3 Index Mode Addressing

MODE	ASSEMBLER NOTATION
Register deferred index	(Rn) [Rx]
Autoincrement indexed	(Rn) + [Rx]
Immediate indexed	I# constant [Rx] which is recognized by assembler but is not generally useful. Operand address is independent of value of constant.
Autoincrement deferred indexed	@(Rn) + [Rx]
Absolute indexed	@#address [Rx]
Autodecrement indexed	−(Rn) [Rx]
Byte, word or longword displacement indexed	B↑D(Rn) [Rx] W↑D(Rn) [Rx] L↑D(Rn) [Rx]
Byte, word or longword displacement deferred indexed	@B↑D(Rn) [Rx] @W↑D(Rn) [Rx] @L↑D(Rn) [Rx]

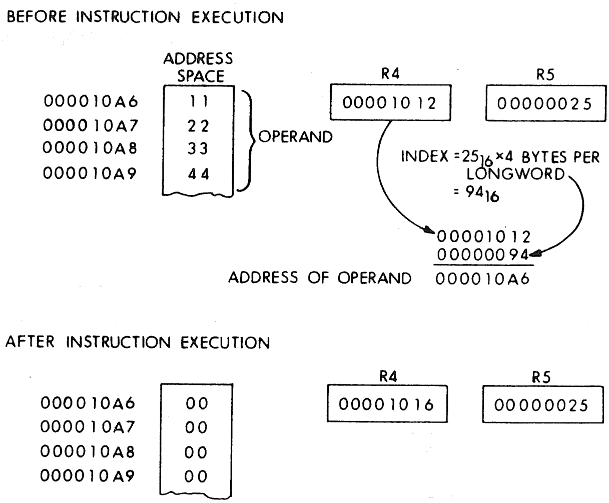
It is important to note that the operand address (the address containing the operand) is first evaluated and then the index specified by the index register is added to the operand address to find the indexed address. To illustrate this, an example of each type of indexed addressing is shown on the following pages.

EXAMPLE: REGISTER DEFERRED INDEXED MODE, INCREMENT WORD INSTRUCTION
INCW (R2) [R5]


BEFORE INSTRUCTION EXECUTION
ADDRESS SPACE
00001012 04
00001013 56
00001014 78
00001015 87
00001018 45
00001019 67
OPERAND
R2
00001012
R5
00000003
3_{16} × 2 BYTES PER WORD = 6
00001012
+6
00001018
AFTER INSTRUCTION EXECUTION
00001018 46
00001019 67
R2
00001012
R5
00000003
ASSEMBLY CODE: ASSUME STARTING LOCATION 00003000
00003000 B6 OPCODE FOR INCREMENT WORD INSTRUCTION
00003001 45 INDEX MODE, REGISTER R5
00003002 62 REGISTER DEFERRED MODE, REGISTER R2

This example shows an Increment Word instruction using register deferred index addressing. The base operand address is evaluated. This location is indexed by six since the value (3) in the index register is multiplied by the word data size of two.

EXAMPLE: AUTOINCREMENT INDEXED MODE, CLEAR LONGWORD INSTRUCTION
CLRL (R4) + [R5]

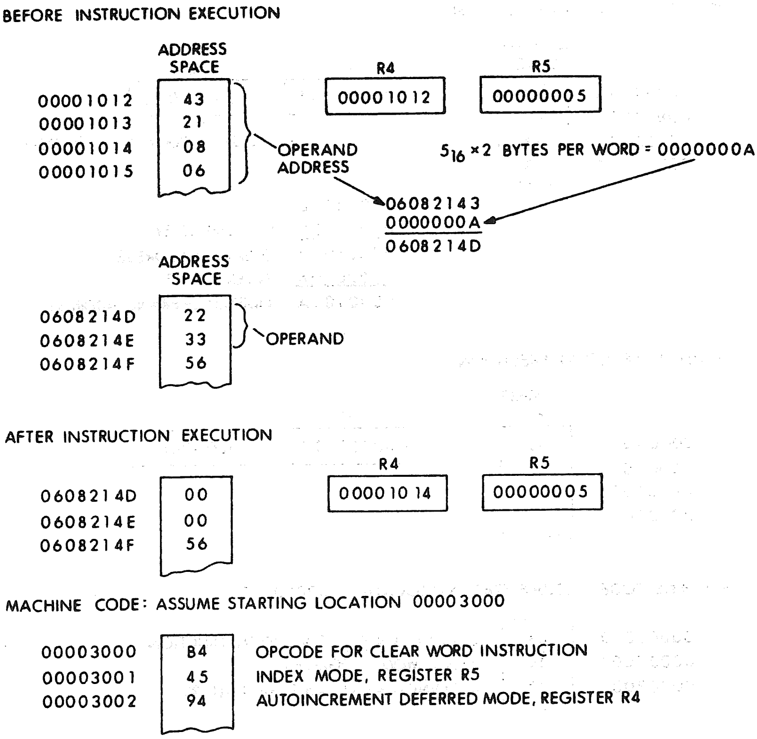


MACHINE CODE: ASSUME STARTING LOCATION 00003000

00003000	D4	OPCODE FOR CLEAR LONGWORD INSTRUCTION
00003001	45	INDEX MODE, REGISTER R5
00003002	84	AUTOINCREMENT MODE, REGISTER R4

This example shows a Clear Long instruction using the autoincrement indexed addressing mode. The base operand address is in R4. This value is indexed by the quantity in R5 muliplied by the data size. This location, plus the next three, are cleared since a clear longword instruction is specified.

EXAMPLE: AUTOINCREMENT DEFERRED INDEX MODE, CLEAR WORD INSTRUCTION
CLRW @(R4) + [R5]



BEFORE INSTRUCTION EXECUTION

	ADDRESS SPACE
00001012	43
00001013	21
00001014	08
00001015	06

R4: 00001012

R5: 00000005

5_{16} × 2 BYTES PER WORD = 0000000A

```
 06082143
 0000000A
 --------
 0608214D
```

	ADDRESS SPACE
0608214D	22
0608214E	33
0608214F	56

AFTER INSTRUCTION EXECUTION

0608214D	00
0608214E	00
0608214F	56

R4: 00001014

R5: 00000005

MACHINE CODE: ASSUME STARTING LOCATION 00003000

00003000	B4	OPCODE FOR CLEAR WORD INSTRUCTION
00003001	45	INDEX MODE, REGISTER R5
00003002	94	AUTOINCREMENT DEFERRED MODE, REGISTER R4

This example shows a Clear Word instruction using the autoincrement deferred indexing mode. R4 contains the address of the operand address. The index value A is obtained by multiplying the contents (5) of the index register by the context of the data type, which is 2. The calculated word address is cleared.

EXAMPLE: AUTODECREMENT INDEXED MODE, CLEAR WORD INSTRUCTION
CLRW#−(R2) [R4]

BEFORE INSTRUCTION EXECUTION

	ADDRESS SPACE
0000101A	33
0000101B	33
0000101C	33
0000101D	33

R2	R4
00001016	00000003

3_{16} × 2 BYTES PER WORD = 6 (INDEX)

00001016	
<u>00000002</u>	DECREMENT BY 2
00001014	OPERAND ADDRESS
<u>00000006</u>	INDEX VALUE
0000101A	INDEXED OPERAND ADDRESS

AFTER INSTRUCTION EXECUTION

	ADDRESS SPACE
0000101A	00
0000101B	00
0000101C	33
0000101D	33

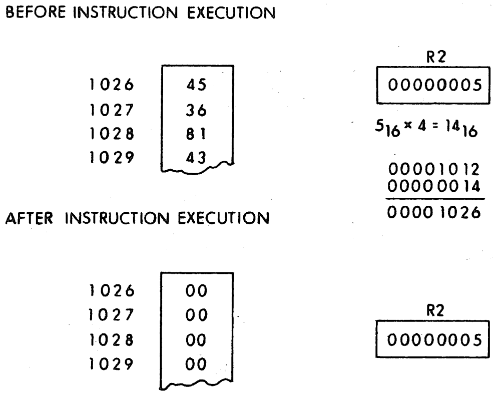
R2	R4
00001014	00000003

MACHINE CODE: ASSUME STARTING LOCATION 00003000

00003000	B4	OPCODE FOR CLEAR WORD INSTRUCTION
00003001	44	INDEX MODE, REGISTER R4
00003002	72	AUTOINCREMENT MODE, REGISTER R2

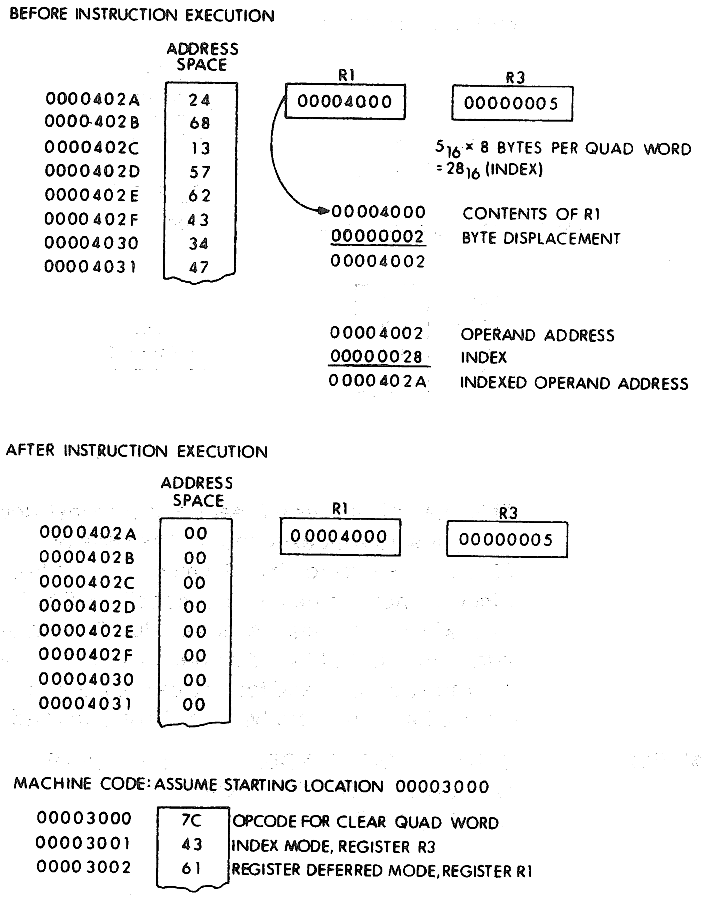
This example shows a Clear Word instruction using autodecrement indexed mode. The content of R2 is predecremented and the indexed value is calculated as six. Since a clear word instruction is specified, two bytes are cleared.

EXAMPLE: ABSOLUTE INDEXED MODE, CLEAR LONGWORD INSTRUCTION
CLRL @ #↑X1012 [R2]



This example shows a Clear Longword instruction using absolute indexed mode. The base of 00001012 is indexed by R2 which contains five. Since a longword data type is specified, 5 X 4 = 14_{16}, which becomes the index value. This value is added to 00001012 yielding 0001026. This is the operand address, and four bytes are cleared since a longword data type has been specified.

EXAMPLE: DISPLACEMENT INDEXED MODE, CLEAR QUADWORD INSTRUCTION
CLRQ 2(R1) [R3]

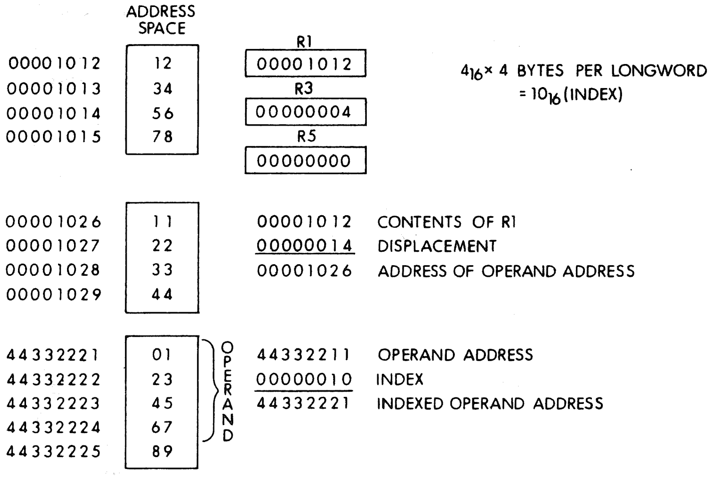


This example shows a Clear Quadword instruction using displacement index mode. The byte displacement of two is added to the content of R1. The index which is calculated as 28 is added to this address. This location and the next seven locations (since a quadword instruction is specified) are cleared.

EXAMPLE: DISPLACEMENT DEFERRED INDEX MODE, MOVE LONG INSTRUCTION
MOVL @ ↑X14 (R1) [R3], R5

This example shows a Move Long instruction using displacement deferred indexed addressing. The displacement of 14 is added to the contents of R1 yielding 00001026. The contents of this location yield the operand address (44332211). This quantity is added to the index yielding the indexed operand address of 44332221. The contents of this address are then moved into R5 as shown.

BEFORE INSTRUCTION EXECUTION



ADDRESS	ADDRESS SPACE
00001012	12
00001013	34
00001014	56
00001015	78
00001026	11
00001027	22
00001028	33
00001029	44
44332221	01
44332222	23
44332223	45
44332224	67
44332225	89

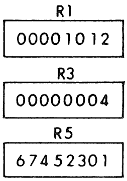
R1	00001012
R3	00000004
R5	00000000

$4_{16} \times 4$ BYTES PER LONGWORD $= 10_{16}$ (INDEX)

00001012	CONTENTS OF R1
00000014	DISPLACEMENT
00001026	ADDRESS OF OPERAND ADDRESS

44332211	OPERAND ADDRESS
00000010	INDEX
44332221	INDEXED OPERAND ADDRESS

AFTER INSTRUCTION EXECUTION



R1	00001012
R3	00000004
R5	67452301

PROGRAM COUNTER ADDRESSING

Register 15 is used as the program counter. It can also be used as a register in addressing modes. The processor increments the program counter as the opcode, operand specifier and immediate data or addresses (of the instruction) are evaluated. The amount that the PC is incremented is determined by the opcode, number of operand specifiers, and so on.

The PC can be used with all of the VAX addressing modes, except register or index mode, since in those two modes the results will be unpredictable. The following modes utilize the PC as the general register.

Mode	**Name**	**Assembler**	**Function**
8	Immediate	I↑#Operand	Constant operand follows address mode
9	Absolute	@#Location	Absolute address follows address mode
A	Byte relative	B↑G (R)	Displacement is added to current value of PC to obtain operand address
C	Word relative	W↑G (R)	
E	Longword relative	L↑G (R)	
B	Byte relative deferred	@B↑G (R)	Displacement is added to current value of PC to yield address of operand address
D	Word relative deferred	@W↑G (R)	
F	Longword relative deferred	@L↑G (R)	

Immediate Mode — same as autoincrement mode, with PC used as general register.

Absolute Mode — same as autoincrement deferred mode, with PC used as general register.

Relative Mode — same as displacement mode, with PC used as general register.

Relative Deferred Mode — same as displacement deferred mode with PC used as general register.

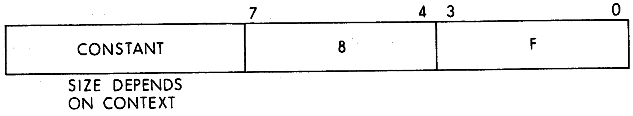
When a standard program is available for different users, it is often helpful to be able to run it at different areas of virtual memory. VAX computers can accomplish the relocation of a program very efficiently through the use of position-independent code (PIC). If an instruction and its objects are moved in such a way that the relative distance between them is not altered, the same offset relative to the PC can be used in all positions in memory.

Immediate Mode

Assembler Syntax: I↑# operand

Mode Specifier: 8

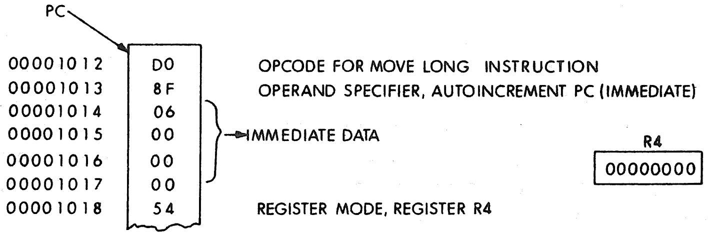
Operand Specifier Format:



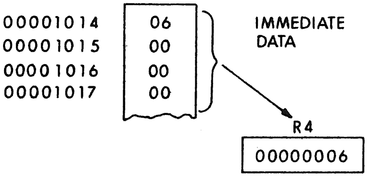
Description: The immediate addressing mode is autoincrement mode when the PC is used as the general register. The contents of the location following the addressing mode are immediate data.

EXAMPLE: IMMEDIATE MODE, MOVE LONG INSTRUCTION
MOVL #6, R4

BEFORE INSTRUCTION EXECUTION


PC
00001012
00001013
00001014
00001015
00001016
00001017
00001018
D0
8F
06
00
00
00
54
OPCODE FOR MOVE LONG INSTRUCTION
OPERAND SPECIFIER, AUTOINCREMENT PC (IMMEDIATE)
IMMEDIATE DATA
REGISTER MODE, REGISTER R4
R4
00000000

AFTER INSTRUCTION EXECUTION



This example shows a Move Long instruction using immediate mode. The immediate data (00000006) following the opcode and operand specifier are moved to the contents of R4.

Absolute Mode

Assembler Syntax: @#location

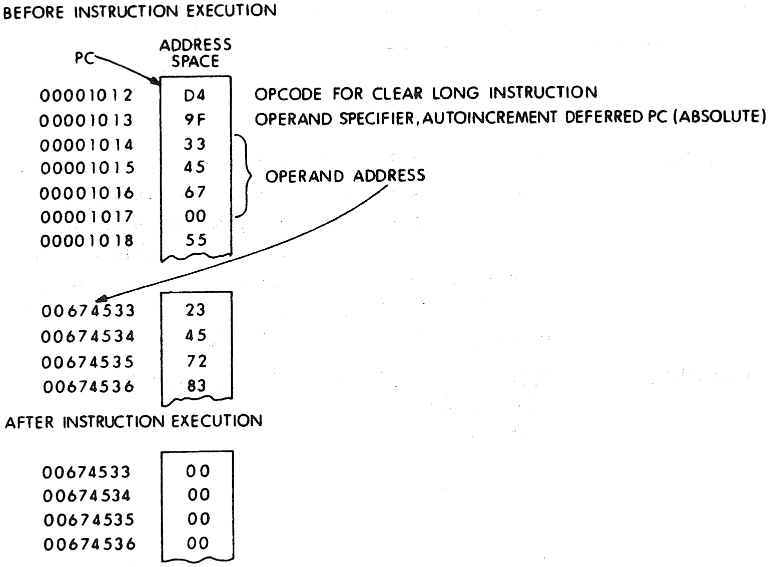
Mode Specifier: 9

Operand Specifier Format:

39 8	7 4	3 0
ADDRESS	9	F

Description: This mode is autoincrement deferred using the PC as the general register. The contents of the location following the addressing mode are taken as the operand address. This is interpreted as an absolute address (an address that remains constant no matter where in memory the assembled instruction is executed).

EXAMPLE: ABSOLUTE MODE, CLEAR LONG INSTRUCTION
CLRL @#↑X674533



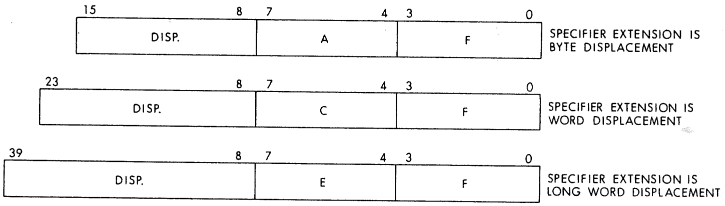
This example shows a Clear Longword instruction using the absolute addressing mode. This instruction causes the location(s) following the addressing mode to be taken as the address of the operand, and is 00674533, in this case. The longword operand associated with this address is cleared.

Relative Mode

Assembler Syntax: B↑D—Byte displacement
W↑D—Word displacement
L↑D—Longword displacement

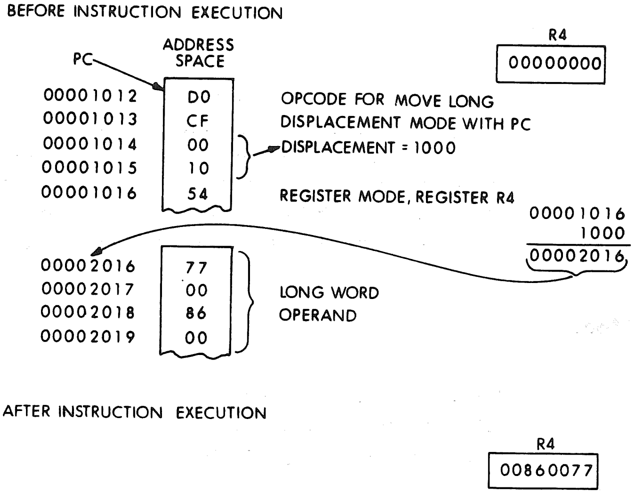
Mode Specifier: A (Byte), C (Word), E (Longword)

Operand Specifier Format:


15
8
7
4
3
0
DISP.
A
F
SPECIFIER EXTENSION IS BYTE DISPLACEMENT
23
8
7
4
3
0
DISP.
C
F
SPECIFIER EXTENSION IS WORD DISPLACEMENT
39
8
7
4
3
0
DISP.
E
F
SPECIFIER EXTENSION IS LONG WORD DISPLACEMENT

Description: This mode is the displacement mode with the PC used as the general register. The displacement, which follows the operand specifier, is added to the PC and the sum becomes the address of the operand. This mode is useful for writing position-independent code, since the location referenced is always fixed relative to the PC.

EXAMPLE: RELATIVE MODE, MOVE LONGWORD INSTRUCTION
MOVL ↑X2016, R4



This example shows a Move Long instruction using relative mode. The word following the address mode is added to the PC to obtain the address of the operand.

In this example, the PC is pointing to location 00001016 after the first operand specifier is evaluated. The word following the opcode and first operand specifier is 00001000, and is added to the PC yielding 00002016. This value represents the address of the longword operand (00860077). This operand is then moved to register R4. The PC contains 00001017 after instruction execution.

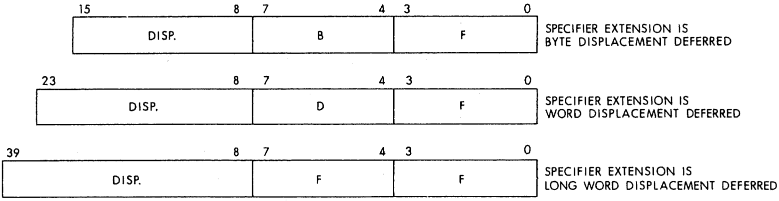
Relative Deferred Mode

Assembler Syntax:

@B↑D—Byte displacement deferred
@W↑D—Word displcement deferred
@L↑D—Longword displacement deferred

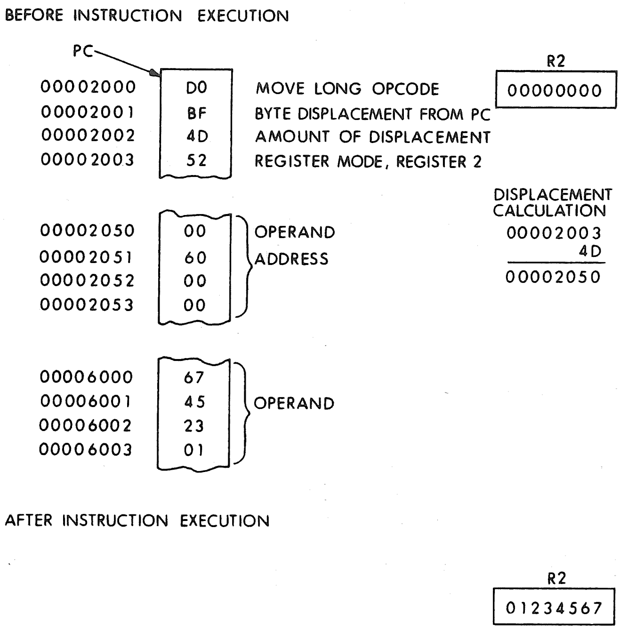
Mode Specifier: B (byte deferred), D (word deferred), F (longword deferred)

Operand Specifier:



Description: This mode is similar to relative mode, except that the displacement, which follows the addressing mode, is added to the PC and the sum is a longword address of the address of the operand. This addressing mode is useful when processing tables of addresses.

EXAMPLE: RELATIVE DEFERRED MODE, MOVE LONG INSTRUCTION
MOVL @↑X2050, R2



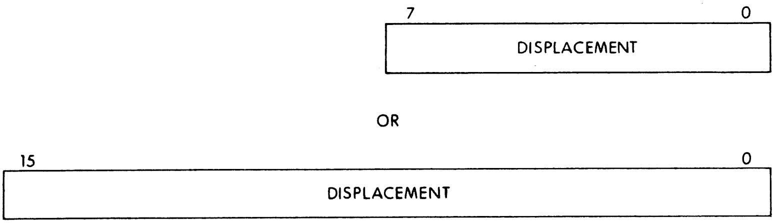
This example shows a Move Long instruction where 00002050 represents the address of the operand. A byte displacement would be selected by the assembler since the displacement is within 128 (decimal) addressable bytes. When the displacement is evaluated, the program counter is pointing to 00002003. The displacement of 4D is added to the current value of the PC yielding the address of 00002050. The contents of this address are then used as the effective operand address of 00006000, and the operand of 1234567 is moved to R2.

Branch Addressing

Assembler Syntax: A

Mode Specifier: None

Operand Specifier Format:



Description:

In branch displacement addressing, the byte or word displacement is sign-extended to 32 bits and added to the updated content of the PC. The updated content of the PC is the address of the first byte beyond the operand specifier.

The assembler notation for byte and word branch displacement addressing is A where A is the branch address. Note that the branch address and not the displacement is used.

Branch instructions are most frequently used after instructions like compare (CMP) and are used to cause different actions depending on the results of that compare.

EXAMPLE:

UNSIGNED BRANCH
This example causes a branch to location NOT if C is not a digit (i.e., C is treated as an unsigned number outside the range 0 through 9).

```
CMPB C, #↑A/0/     ;Compare C and ASCII representation of digit 0.

BLSSU NOT          ;Branch to locaton NOT if less
                   than an unsigned 0.

CMPB C, #↑A/9/     ;Compare C and ASCII representation of digit 9.

BGTRU NOT          ;Branch to locaton NOT if greater
                   than an unsigned 9.
```

EXAMPLE: BRANCH ON BIT

```
BBS #2,B,X          ;branches to X if bit #2 in B
                    ;is set (= 1)

BBSC #2,B,X         ;branches to X if
                    ;bit #2 in B is set (= 1) and
                    ;bit is then cleared

BLBS B,X            ;branches to X if bit
                    ;0 of B is set (= 1)
```

DEC Datasystem

CHAPTER 6

MEMORY, REGISTERS, AND PROCESSOR STATUS—AN OVERVIEW

INTRODUCTION

VAX architecture is intended to support multiprogramming, the concurrent execution of a number of processes in a single computer system. (A **process** can be defined for now as a single stream of machine instructions executed in sequence.)

Virtual address space is mapped into the physical address space by the processor's memory management logic. In addition, the memory management hardware supports **paging**, a technique by which the system keeps in physical memory only those parts of a process's virtual memory actively in use.

A VAX process exists in and operates on a memory space of 2^{32} bytes; certain addresses and data are kept in the sixteen 32-bit general registers; and a small number of processor state variables are kept in a special register called the Processor Status Longword, or PSL. The combined set of information in memory, general registers, and PSL actually defines a process. The rest of this chapter will detail these components of the process.

A reminder: if a term unfamiliar to you appears below, please check the Glossary and the Index for details.

MEMORY

One half of the virtual address space (that with the most significant bit set) is referred to as **system space**, because it is the same for all processes in the system. System space contains the operating system software and systemwide data, and to facilitate interrupt handling and system service routines it is shared by all processes.

The other half of the virtual address space (that with the most significant address bit cleared) is separately defined for each process; it is therefore referred to as **process space** (or sometimes, per-process space). Process space is further subdivided (on the next most significant address bit) into P0 space, in which program images and most of their data reside; and P1 space, in which the system allocates space for stacks and process-specific data. Because P1 space is used for stacks, which grow toward lower addresses, it is unique in that it is allocated from high addresses downward. P0 and P1 space together constitute a process's working memory. Except for special cases of sharing, each process has its own P0 and P1 spaces, independent of others in the system. Figure 6-1 illustrates the address spaces of several processes in a multiprogramming system. Each process space is independent of the others, while the system space is shared by all.

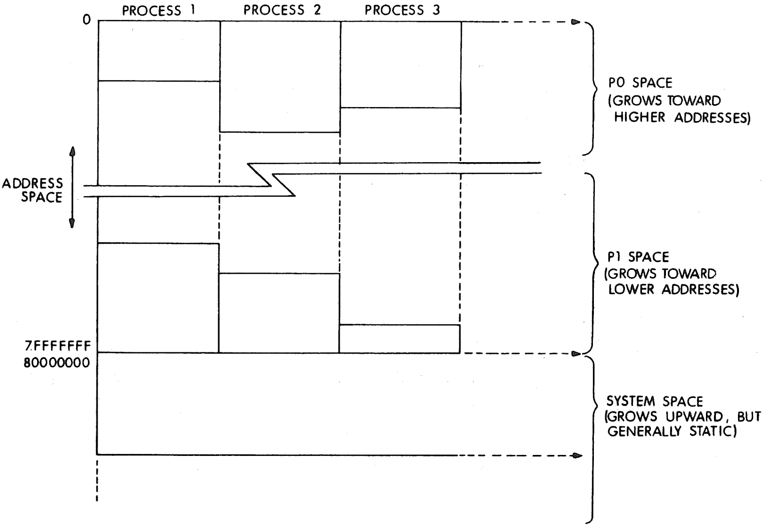


Figure 6-1 Address Spaces in Process Context

Though the basic addressable unit in VAX is the 8-bit byte, larger units can be constructed by doubling byte sizes: a word is two bytes; a longword is four bytes; a quadword is eight bytes; and an octaword is sixteen bytes. These five are the units in which VAX memory stores data, but the processor sometimes interprets operands in other units, such as half bytes (nibbles) for decimal digits, or variable-sized bit fields.

In general, the memory system processes requests only for naturally aligned data. In other words, a byte can be obtained from any address, but a word can only come from an even address, a longword can only come from an address which is a multiple of four, and so on. VAX processors have a provision for converting an unaligned request into a sequence of requests that can be accepted by the memory; however, this conversion can have a serious impact on performance, and data structures *should* be designed in such a way that the natural alignment of operands is preserved wherever possible.

The VAX memory management logic serves six principle purposes.

1. A number of processes may occupy main memory simultaneously, all freely using process space addresses, while referring independently to their own programs and data.
2. The operating system keeps selected parts of a process and its data in memory, bringing in other parts as needed, without explicit intervention by the program. Large programs can be run in reduced memory space without recoding or overlays visible to the programmer.
3. The operating system may scatter pieces of programs and data wherever space is available in memory, without regard to the apparent contiguity of the program. It is never necessary for the system to shuffle memory in order to collect contiguous space for another process to be brought into memory.
4. Cooperating processes share memory in a controlled way. Two or more processes may communicate through shared memory in which both have read/write access. One process may be granted read access to memory being modified by others; or a number of processes may share a single copy of a read-only area.
5. The operating system limits access to memory according to a privilege hierarchy. Thus, within any address space, privileged software can maintain data bases which it can access, but which code running in less privileged modes cannot.
6. The operating system may grant or inhibit access to control, status, and data registers in peripheral devices and their controllers. Since those registers are part of the physical address space, access to them is achieved by creation of a page table entry (described below) whose page frame number field selects the desired device or controller address in the I/O portion of the physical address space. References to the registers are then under control of the access control field of the page table entry. Thus the same privilege mechanisms which control access to sensitive data in memory are used to control access to I/O devices.

For the purposes of memory management (specifically protection and translation of virtual to physical addresses) the unit of memory is the 512-byte **page**. Pages are always naturally aligned (i.e., the address of the first byte of a page is a multiple of 512). Virtual addresses are 32 bits long, and are partitioned by the memory management logic as shown in Figure 6-2.

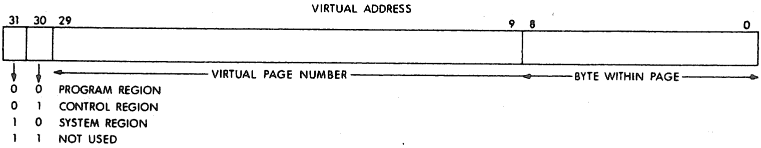


Figure 6-2 Virtual Address Format

The nine low-order bits select a byte within a page, and are unchanged by the address translation process. The two high-order bits select the P0, P1, or system portion of the address space. The remaining 21 bits are used to obtain a longword called the **Page Table Entry** (PTE) from the P0, P1, or system page table as appropriate.

The PTE contains four pieces of information:

- Protection code, specifying which, if any, access modes are to be permitted read or write access to the page
- Page frame number, identifying the 512-byte page of physical memory to be used on references to the virtual address
- Valid bit, indicating that the page frame number is valid (i.e., it identifies a page in memory, rather than one in the swapping space on a disk)
- Modification flag, set by the processor whenever a write to the page occurs

Figure 6-3 illustrates the Page table entry format.

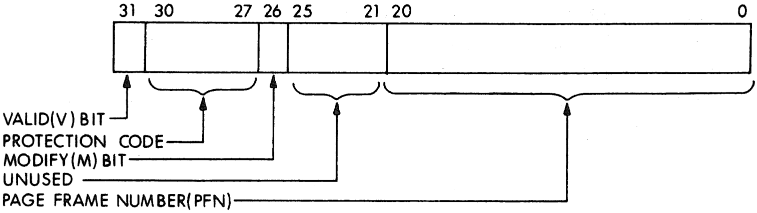


Figure 6-3 Page Table Entry Format

In concept, the process of obtaining a page table entry occurs on every memory reference. In practice, however, the processor maintains a translation buffer, a special-purpose cache of recently used

PTEs. Most of the time, the translation buffer already contains the PTEs for the virtual addresses used by the program, and the processor does not need to go to memory to obtain them.

There is one page table entry for each existing page of the virtual address space. A length register associated with each region specifies how many pages exist in that region of the address space. The System Page Table (SPT), which contains PTEs for addresses greater than 80000000_{16}, is allocated to contiguous pages in physical memory. Since the size of system space is relatively constant and can be determined at system startup time, allocating a fixed amount of physical memory to the SPT poses no problems.

Process space page tables, on the other hand, change quite dynamically and can become very large. Because it would be awkward for the operating system to have to keep the process page tables in contiguous areas of physical memory, VAX defines structures called the **process space page tables**, P0PT and P1PT. P0PT and P1PT are to be allocated in contiguous areas of system space (i.e., virtual memory). Thus, the mapping for process space addresses involves two memory references—one to translate the process space address into a physical memory address, and the second to translate the system virtual address of the table containing the first translation. It is important to notice that even if the translation buffer does not have the mapping for the process space address, it *is* likely to have that for the page table, and thus can save one of the references.

PROCESSOR STATUS LONGWORD

There are several processor state variables associated with each process, and VAX groups them together into the 32-bit Processor Status Longword (or **PSL**). Bits 15:0 of the PSL are referred to separately as the Processor Status Word (PSW). The PSW contains unprivileged information, and those bits of the PSW which have defined meaning are freely controllable by any program. Bits 31:16 of the PSL have privileged status, and while any program can perform the REI instruction (which loads PSL), REI will refuse to load any PSL which would increase the privilege of a process, or create an undefined state in the processor. Figure 6-4 illustrates the Processor Status Longword, and the following paragraphs explain the various fields.

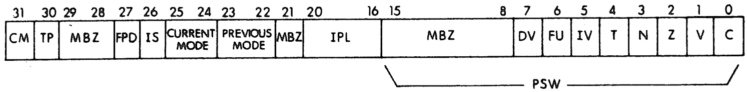


Figure 6-4 Processor Status Longword

Bits 3:0 of the PSL are termed the condition codes; in general they reflect the result status of the most recent instruction which affects them. The condition codes are tested by the conditional branch instructions.

N Bit—Bit 3 is the **N**egative condition code; in general it is set by instructions in which the result stored is negative, and cleared by instructions in which the result stored is positive or zero. For those instructions which affect N according to a stored result, N reflects the actual result, even if the sign of the result is algebraically incorrect as a result of overflow.

Z Bit—Bit 2 is the **Z**ero condition code; typically it is set by instructions which store a result that is exactly zero, and cleared if the result is not zero. Again, this reflects the actual result, even if overflow occurs.

V Bit—Bit 1 is the o**V**erflow condition code; in general it is set after arithmetic operations in which the magnitude of the algebraically correct result is too large to be represented in the available space, and cleared after operations whose result fits. Instructions in which overflow is impossible or meaningless either clear V or leave it unaffected. Note that all overflow conditions which set V can also cause traps if the appropriate trap enable bits are set.

C Bit—Bit 0 is the **C**arry condition code; usually it is set after arithmetic operations in which a carry out of, or borrow into, the most significant bit occurred. C is cleared after arithmetic operations which had no carry or borrow, and either cleared or unaffected by other instructions. The C bit is unique in that it not only determines the operation of conditional branch instructions, it also serves as an input variable to the ADWC (Add with Carry) and SBWC (Subtract with Carry) instructions used to implement multiple-precision arithmetic.

Bits 7:4 of the PSL are trap-enable flags, which cause traps to occur under special circumstances.

DV Bit—Bit 7 is the **D**ecimal o**V**erflow trap enable. When set, it causes a decimal overflow trap after the execution of any instruction which produces a decimal result whose absolute value is too large to be represented in the destination space provided. When DV is clear, no decimal overflow trap occurs. The result stored consists of the low-order digits and sign of the algebraically correct result. Note that there are other trap conditions for which there are no enable flags—division by zero and floating overflow.

FU Bit—Bit 6 is the **F**loating **U**nderflow trap enable. When set, it causes a floating underflow trap after the execution of any instruction which produced a floating result too small in magnitude to be represented. When FU is clear, no floating underflow trap occurs. The result stored is zero when floating underflow occurs, regardless of the state of FU.

IV Bit—Bit 5 is the **I**nteger o**V**erflow trap enable; when set, it causes an integer overflow trap after an instruction which produced an integer result that could not be correctly represented in the space provided. When bit 5 is clear, no integer overflow trap occurs. The V condition code is set independently of the state of IV (bit 5).

T Bit—Bit 4 is the **T**race bit; when set, it causes a trace trap to occur after execution of the next instruction. This facility is used by debugging and performance analysis software to step through a program one instruction at a time. If any instruction is traced and causes an arithmetic trap, the trace trap occurs after the arithmetic trap.

Bits 15:8 of the PSL are unused, and reserved.

IPL Bits—Bits 20:16 represent the processor's **I**nterrupt **P**riority **L**evel. An interrupt, in order to be acknowledged by the processor, must be at a priority higher than the current IPL. Virtually all software runs at IPL 0, so the processor acknowledges and services interrupt requests of any priority. The interrupt service routine for any request, however, runs at the IPL of the request, thereby temporarily blocking interrupt requests of lower or equal priority. Briefly, there are 31 priority levels above zero, numbered in hexadecimal 01 through 1F. Interrupt levels 01_{16} through $0F_{16}$ exist entirely for use by software. Levels 10_{16} through 17_{16} are for use by peripheral devices and their controllers, though present systems support only 14_{16} through 17_{16}. Levels 18_{16} to $1F_{16}$ are for use for urgent conditions, including the interval clock, serious errors, and power fail.

Bit 21—Must be zero.

Previous Mode Bits—Bits 23:22 are the previous mode field, which contains the value from the current mode field at the most recent exception which transferred from a less privileged mode to this one. Previous mode is of interest, for example, in the PROBE instructions, which enable privileged routines to determine whether a caller at the previous mode is sufficiently privileged to reference a given area of memory.

Current Mode Bits—Bits 25:24 are the current mode field, which determines the privilege level of the currently executing program.

Privilege is granted in two ways by the mode field—certain instructions (HALT, Move To Processor Register, and Move From Processor Register) are not performed unless the current mode is kernel. The memory management logic controls access to virtual addresses on the basis of the program's current mode, the type of reference (read or write), and a protection code assigned to each page of the address space.

IS Bit—Bit 26 is the Interrupt Stack flag, which indicates that the processor is using the special "interrupt stack" rather than one of the four stacks associated with the current process. When IS is set, the current mode is always kernel; thus, software operating "on the interrupt stack" has full kernel mode privileges.

FPD Bit—Bit 27 is the First Part Done flag, which the processor uses in certain instructions which may be interrupted or page faulted in the middle of their execution.

If FPD is set when the processor returns from an exception or interrupt, it resumes the interrupted operation where it left off, rather than restarting the instruction.

TP Bit—Bit 30 is the Trace Pending bit, which is used by the processor to ensure that one, and only one, trace trap occurs for each instruction performed with the Trace bit (bit 4) set.

CM Bit—Bit 31 is the Compatibility Mode bit. When CM is set, the processor is in PDP-11 compatibility mode, and executes PDP-11 instructions. When CM is clear, the processor is in native mode, and executes VAX instructions.

GENERAL REGISTERS

VAX provides sixteen general registers for temporary address and data storage. Registers are denoted either **Rn** or **R[n]**, where n is an integer in the range 0 through 15. Registers do not have memory addresses, but are accessed either explicitly by inclusion of the register number in an operand specifier, or implicitly by machine operations which make reference to specific registers. Certain registers have specific uses and special names:

PC — R15 is the Program Counter (PC). The processor updates it to address the next byte of the program; therefore, PC is not used as a temporary, accumulator, or index register.

SP — R14 is the Stack Pointer (SP). Several instructions make implicit references to SP, and most software assumes that SP points to memory set aside for use as a stack. There is no restriction on the explicit use of other registers (except PC) as stack pointers, though those instructions which make implicit references to the stack always use SP.

FP

R13 is the Frame Pointer (FP). The VAX procedure call convention builds a data structure on the stack called a stack frame. The CALL instructions load FP with the base address of the stack frame, and the RETurn instruction depends on FP's containing the address of a stack frame. Further, VAX software depends on maintenance of FP for correct reporting of certain exceptional conditions.

AP

R12 is the Argument Pointer (AP). The VAX procedure call convention uses a data structure called an argument list, and needs AP as the base address of the argument list. The CALL instructions load AP in accordance with that convention, but there is no hardware or software restriction on the use of AP for other purposes.

R6:R11

Registers 6 through 11 have no special significance either to hardware or the operating system. Specific software will assign specific uses for each register.

R0:R5

Registers 0 through 5 are generally available for any use by software, but are also loaded with specific values by those instructions whose execution must be interruptable—the character string, decimal arithmetic, Cyclic Redundancy Check, and Polynomial instructions. The specific instruction descriptions identify which registers are used, and what values are loaded into them.

As you can see, the general philosopy of DIGITAL software governing the allocation of registers is that high-numbered registers should have the most global significance, and low-numbered registers are used for the most temporary, local purposes. While there is no technical basis for this rule, it is a matter of convention followed by both hardware and system software. Thus, high-numbered registers are used for pointers needed by all software and hardware, and low-numbered registers are used for the working storage of string-type instructions. Similarly, the VAX procedure call convention regards R0 and R1 as so temporary that they are not even saved on calls. This is because R0 and R1 are used to return function values. Table 6-1 summarizes the hardware and conventional software use of the general registers.

Table 6-1 Special Register Usage

Registers	Hardware Use	Conventional Software Use
PC (R15)	Program counter	Program counter
SP (R14)	Stack pointer	Stack pointer
FP (R13)	Frame pointer saved & loaded by CALL, used & restored by RET	Frame pointer; condition signalling
AP (R12)	Argument pointer saved & loaded by CALL, restored by RET	Argument pointer (base address of argument list)
R6:R11	None	Any
R3, R5	Address counter in character & decimal instructions	Any
R2, R4	Length counter in character & decimal instructions	Any
R1	Result of POLYD; address counter in character & decimal instructions	Result of functions (not saved or restored on procedure call)
R0	Results of POLY, CRC; length counter in character & decimal instructions	Results of functions, status of services (not saved or restored on procedure call)

STACKS

Stacks, also called pushdown lists or last-in/first-out queues, are an important feature of the architecture. They are used for:

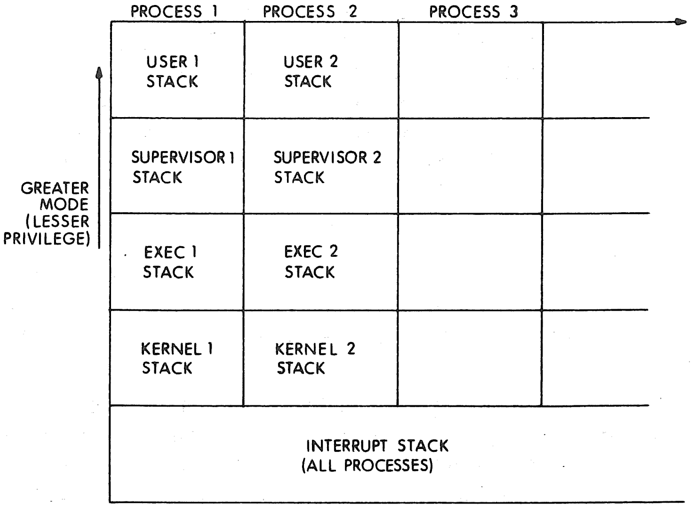
- Saving the general registers, including PC, at entry to a subroutine, for restoration at exit
- Saving PC, PSL, and general registers at the time of interrupts and exceptions, and during context switches
- Creating storage space for temporary use or for nesting of recursive routines

A stack is implemented in VAX by a block of memory and a general register which addresses the "top" of the stack. The "top" of the stack is that location in the block which contains the next candidate for removal. An item is added to the stack ("pushed on") by decrementing the register which serves as the stack pointer, and storing the item at the address in the updated register. The pointer is decremented by the length of the item added to the stack, to allow enough room for it. Conversely, the top item is removed ("popped off") by adding the length of the item to the stack pointer after the last use of the item. These operations are built into the basic addressing mechanisms of VAX instructions; thus, any instruction can operate on the stack, and it is seldom necessary to devote separate instructions to maintenance of the stack pointer.

A stack is usually bounded by inaccessible pages, in order to catch the common programming errors associated with stacks: pushing on more data than there is space to store and popping off more than was pushed. By placing the stack in a block of memory between inaccessible pages, the programmer can be confident of finding such errors. The operating system initializes the stacks this way.

Many VAX processor operations make use of the stack implicitly (i.e., without explicit specification of SP in an operand specifier). This occurs in instructions used in calling and returning from subroutines, and in the processor sequences which initiate and terminate interrupt or exception service routines. In all such cases, the processor uses the stack addressed by R14.

This does not mean that exceptions, interrupts, and system services are performed on the same stacks employed by user-mode programs. The processor maintains five internal registers as pointers to separate blocks of memory to be used as stacks, and uses one or another as SP depending on the current access mode and interrupt stack bit in the processor status longword. Whenever the current access mode and/or interrupt stack bits change, the processor saves the contents of SP into the internal register selected by the old value of those bits, and loads SP from the register selected by the new value. There is one interrupt stack for the entire system, but the kernel, executive, supervisor, and user mode stacks are different for each process in the system. Figure 6-5 illustrates the relationships of the five stacks and multiple processes.



PROCESS 1	PROCESS 2	PROCESS 3	
USER 1 STACK	USER 2 STACK		
SUPERVISOR 1 STACK	SUPERVISOR 2 STACK		
EXEC 1 STACK	EXEC 2 STACK		
KERNEL 1 STACK	KERNEL 2 STACK		
INTERRUPT STACK (ALL PROCESSES)			

Figure 6-5 Stacks by Mode vs. Processes

This multiple-stack mechanism offers a number of advantages over a single stack.

- User-mode programs are not subject to sudden and nonreproducible changes in the data beyond the end of their stack. While it is bad practice to depend on such data, it would also be poor design to make it difficult to debug programs which did depend on such data, either intentionally or through programming error.
- The integrity of a privileged mode program cannot be compromised by a less privileged caller. Even if the caller has completely filled its own stack, the privileged code is in no danger of running out of space, because separate blocks of memory are allocated to the stack associated with each mode.
- Privileged mode programs are not vulnerable to accidental (or malicious) destruction of the stack pointer by less privileged programs. Even if the user program uses SP as a floating point accumulator, privileged code can still depend on it as a stack pointer, because the processor saves the floating point value and loads the pointer value when a mode change occurs.

- By allocating separate stacks for each mode, VAX-11 can dynamically page most stack space, while ensuring the availability of space for interrupt and page fault service. Interrupt service routines and the page fault handler may be invoked at any time, and must have a small amount of stack available immediately, without waiting for it to be paged in. User programs, on the other hand, may need very large stack spaces, making it desirable to page out those regions which are not in active use. Only the kernel and interrupt stacks need to be resident.

CHAPTER 7
MEMORY MANAGEMENT

INTRODUCTION

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

In other words, memory management is the scheme that provides both memory protection and memory mapping mechanisms for VAX systems. The scheme has been designed to achieve the following goals:

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

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

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

VIRTUAL ADDRESS SPACE

The address space seen by the programmer is a linear array of over 4 gigabytes. It is divided into a collection of 512-byte units called *pages*. The page is the basic unit of both relocation and protection.

Since this virtual address space is too large to be contained in any currently manufactured main memory, memory management provides the mechanism to map the active part of the virtual address space to the available physical address space. It also provides page protection between processes. The operating system controls the memory management tables that map virtual addresses into physical memory addresses. Inactive, but used, parts of the virtual address space are mapped onto external storage media via the operating system. Figure 7-1 is a schematic of virtual address space.

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

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

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

Page Protection

Independent of its location in the virtual address space, a page may be protected according to its use. Even though all of the system space is shared, in that your program may generate any address, the program may be prevented from modifying, or even accessing portions of the

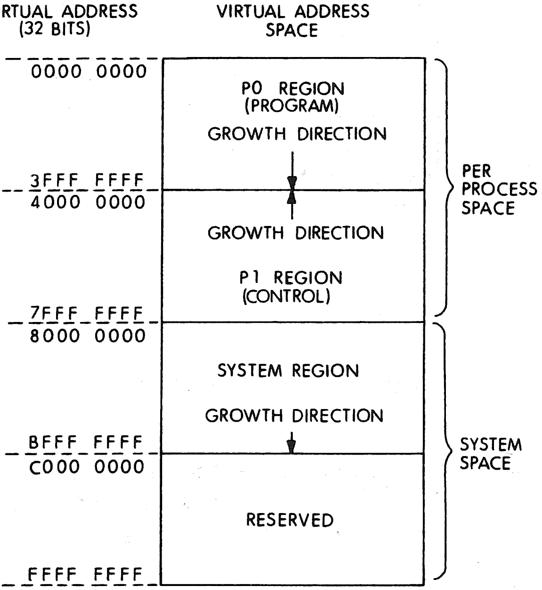


Figure 7-1 Virtual Address Space

system space. A program may also be prevented from accessing or modifying portions of process space.

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

Virtual Address

In order to reference each instruction and operand in memory, the processor generates a 32-bit virtual address as illustrated in Figure 7-2.

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

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

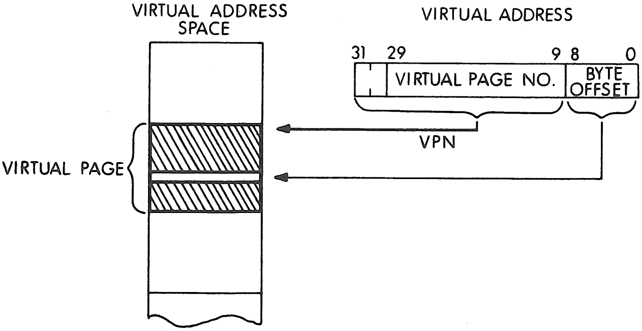


Figure 7-2 Virtual Address

Within the process space, bit 30 distinguishes between the program and control regions. When bit 30 is one, the control region is referenced, and when it is zero, the program region is referenced.

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

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

ACCESS CONTROL

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

Mode

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

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

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

2 Supervisor. Used for such services as command interpretation.

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

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

Protection Code

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

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

This results in 15 possibilities. The protection code is encoded in a 4-bit field in the Page Table Entry described in Table 7-1.

Table 7-1 Protection Codes

CODE		MNEMONIC					
DECIMAL	BINARY		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	

CODE		MNEMONIC					
DECIMAL	BINARY		K	E	S	U	COMMENT

- = 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 7-1.

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);
```

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

ADDRESS TRANSLATION

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

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

The intended access is read if the operation to be performed is a read. The intended access is write if the operation to be performed is a write. If, however, the operation to be performed is a modify (i.e. read 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 7-3.

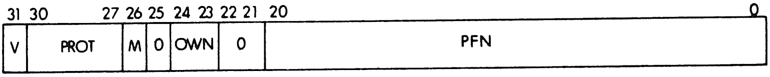


Figure 7-3 Page Table Entry

Bit: 31 **Name:** Valid bit (V)
Function: Governs the validity of the M modify (M) bit and the page frame number (PFN) field. V = 1 for valid; V = 0 for not valid.

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

Bit: 26 **Name:** Modify bit (M)
Function: Set (i.e., = 1) if page has already been recorded as modified. M = 0 if page has not been recorded as modified. Used by

hardware only if V = 1. Hardware sets this bit on a valid, access-allowed memory access associated with a modify or write access, and optionally on a PROBEW or implied probe-write. If a write or modify reference crosses a page boundary and one page faults, it is unpredictable whether the page table entry M bit for the other page is set before the fault. It is unpredictable whether the modification of a process PTE M bit causes modification of the system PTE that maps that process page table. (Note that the update of the M bit is not interlocked in customer-designed multiprocessor systems.)

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

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

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

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

Protection Check Before Valid Check

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

SYSTEM SPACE ADDRESS TRANSLATION

A virtual address with <31:30> = 2 resides in the system virtual address space as illustrated in Figure 7-4.

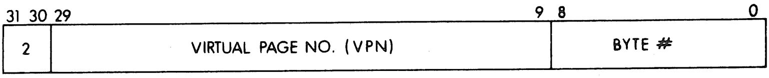


Figure 7-4 System Space Address

The system virtual address space is defined by the system page table (SPT), which is a vector of page table entries (PTEs). The physical base address of the SPT is contained in the System Base Register (SBR). The size of the SPT in longwords, i.e., the number of PTEs, is contained in the System Length Register (SLR). The PTE addressed by the SBR maps the first page of system space, i.e., virtual byte address 80000000_{16}.

The virtual page number is bits <29:9> of the virtual address. Thus, there could be as many as 2^{21} physical pages in the system region. (Typically the value is in the range of a few hundred to a few thousand system pages.) A 22-bit length field is required to express the values 0 through 2^{21} inclusive. At bootstrap time, the content of both registers are unpredictable. The translation from system virtual address to physical address is illustrated in Figure 7-5.

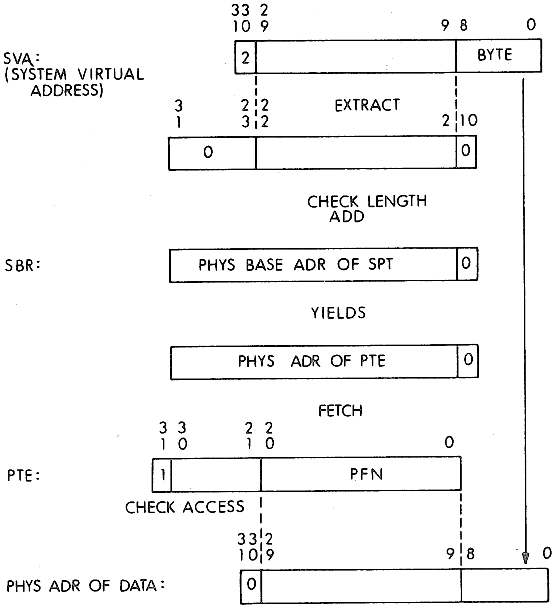


Figure 7-5 System Virtual To Physical Translation

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 one, the P1 region is selected.

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

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

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

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

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

P0 Space

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

The virtual page number is bits <29:9> of the virtual address. Thus, there could be as many as 2^{21} pages in the P0 region. A 22-bit length field is required to express the values 0 through 2^{21} inclusive. P0LR<26:24> are ignored on the Move to Processor Register (MTPR) instruction and read back zero on the Move From Processor Register (MFPR) instruction. At bootstrap time, the contents of both registers are unpredictable. An attempt to load P0BR with a value less than 2^{31} results in a reserved operand fault. The translation from P0 virtual address to physical address is illustrated in Figure 7-6.

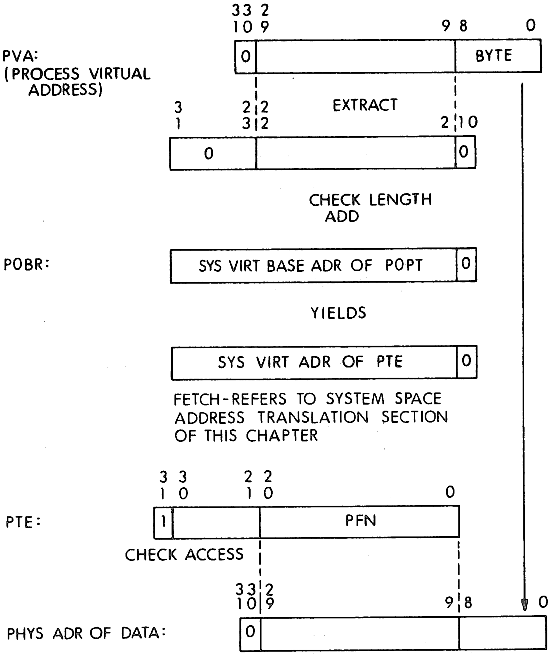


Figure 7-6 P0 Virtual to Physical Translation

P1 Space

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

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

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

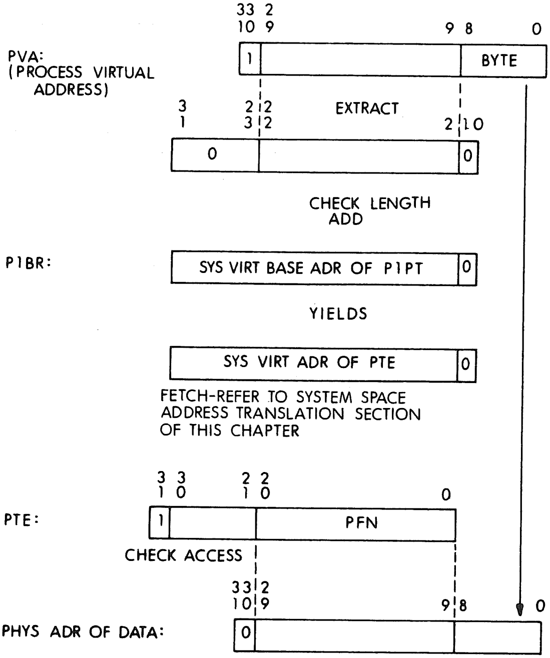


Figure 7-7 P1 Virtual To Physical Translation

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 a value of 0 or 1 according to whether memory management is disabled or enabled, respectively. At bootstrap time, this register is initialized to zero.

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

Translation Buffer

In order to save actual memory references when repeatedly referencing pages, the hardware includes a mechanism, called a *translation buffer,* to remember successful virtual address translations and page status.

Whenever the process context is loaded by the LDPCTX instruction, 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).

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

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 former 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 7-8.

0	M	P	L	:(SP)
SOME VIRTUAL ADDRESS IN THE FAULTING PAGE				
PC OF FAULTING INSTRUCTION				
PSL AT TIME OF FAULT				

Figure 7-8 Fault Parameter Word

The same parameters are stored for both types of faults. The first parameter pushed on the kernel stack after the PSL and PC is the initial virtual address that caused the fault. A process space reference can result in a system space virtual reference for the PTE. If the PTE reference faults, the virtual address that is saved is the process virtual address. In addition, a bit is stored in the fault parameter word indicating that the fault occurred in the PTE reference.

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

Bit: 2 **Name:** Write or Modify Intent
Function: Set to 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.

Bit: 1 **Name:** PTE Reference
Function: 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.

Bit: 0 **Name:** Length Violation
Function: 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.

PRIVILEGED SERVICES AND ARGUMENT VALIDATION

Change Modes

There are four instructions available to allow a program to change the mode at which it is running to a more privileged mode and transfer control to a service dispatcher for the new mode. (Refer to Chapter 10 greater detail.)

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

Validating Address Arguments

Two instructions are provided to allow privileged services to check addresses passed as parameters. To avoid protection holes in the system, a service routine must always validate that its less privileged caller could have directly referenced the addresses passed as parameters. For detail on PROBE instructions, which supply this validation, see Chapter 10.

SHARING

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

Shared Sections in Process Space

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

1. A share count is maintained for each shared page in memory and in effect counts the number of direct pointers to that page.
2. When a process releases a page from its working set and it is a shared page as indicated in the working set data base, the private PTE must be changed to point to the shared PTE for the page, and the private copy of the modify bit must be OR'ed into the shared 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.

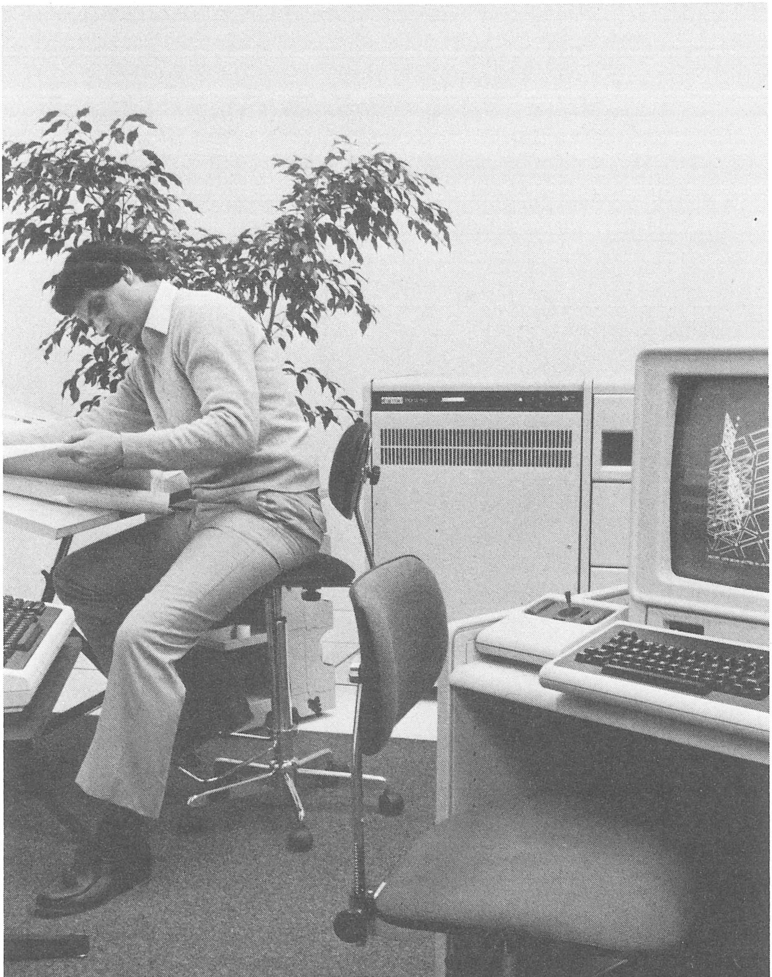
4. Note that throwing a process out of the balance set—the set of all process working sets currently resident in physical memory—is equivalent to releasing all its pages.
5. The use of the indirect page pointer as a software-only mechanism is adequate for this form of sharing. It should be noted that it is very difficult to change the PFN of a page in memory when it is actively being shared. That would require a scan of the page tables for all the processes in the balance set.

Shared Sections in System Space

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

1. Have programs explicitly declare their intention to use each shared system section. This can be done statically at compile or link time or dynamically at run time.
2. Have the balance set manager swap in and lock down the entire section when the process intending to use it is swapped in.
3. The balance set manager maintains share counts on the section and only discards its pages when no process in the balance set wants it.
4. If a process faults such a page because it failed to declare its intention to use the section, then that is a programming error.

Another approach for shared system sections allows a process to reference pages of the section with no prior declaration of its intent to use them. Such pages would be demand paged within a pool of pages reserved for that purpose. That pool would keep a list of the pages in use and a fault for a new one would cause one in the pool to be replaced. This would use the same sort of working set management that is used for the process address space but it would be global across processes.



CHAPTER 8
PROCESS STRUCTURE

DEFINITION OF A PROCESS

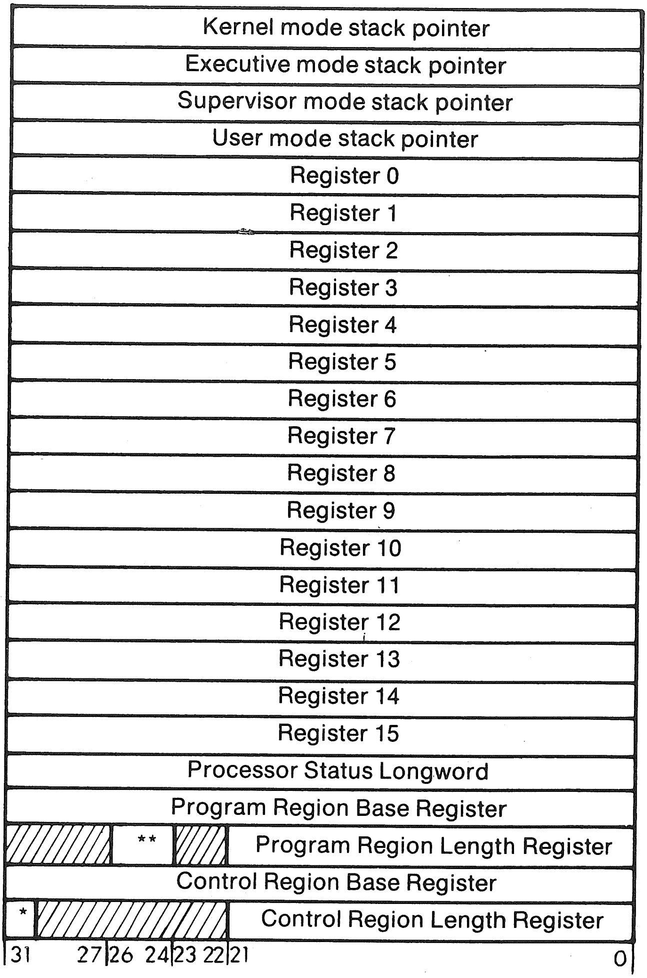
To recap part of Chapter 2, a *process* is the basic entity that may be scheduled for execution by VAX family processors. It consists of an address space, a hardware context, and a software context. The hardware context is defined by a data structure called the process control block (PCB), which contains images of the 14 general purpose registers, the Processor Status Longword (PSL), the Program Counter (PC), the four per-process stack pointers, the process virtual memory defined by the base and length registers (P0BR, P0LR, P1BR, and P1LR), and several minor control fields. When a process is not executing, its hardware context is stored in the process control block. 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.

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

The process control block for the currently executing process is pointed to by the contents of the process control block base (PCBB) register, an internal privileged register, which contains the physical longword address of the PCB.

The PCB itself 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 8-1.


Kernel mode stack pointer
Executive mode stack pointer
Supervisor mode stack pointer
User mode stack pointer
Register 0
Register 1
Register 2
Register 3
Register 4
Register 5
Register 6
Register 7
Register 8
Register 9
Register 10
Register 11
Register 12
Register 13
Register 14
Register 15
Processor Status Longword
Program Region Base Register
**
Program Region Length Register
Control Region Base Register
*
Control Region Length Register
31
27
26
24
23
22
21
0

*Enable performance monitor

**Asynchronous System Trap pending

Figure 8-1 Hardware Process Control Block

A description of the process control block follows.

Longword	**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 0 and Interrupt Stack (IS) is 0.
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,	General registers 0 through 11,
		AP,FP	Argument Pointer, and Frame Pointer.
18	<31:0>	PC	Program Counter.
19	<31:0>	PSL	Processor Status Longword.
20	<31:0>	P0BR	Base register for page table describing process virtual addresses from 0 to $2^{30}-1$.
21	<21:0>	P0LR	Length register for page table located by P0BR. Describes effective length of page table.
	<23:22>	MBZ	Must be 0.

Longword	Bits	Mnemonic	Description
	<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 are discussed below.) 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

Longword	Bits	Mnemonic	Description
	<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.
	<30:22>	MBZ	Must be zero.
	<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 mnemonics shown above.

A process must be executing in kernel mode to alter its P0BR, P1BR, P0LR, P1LR, ASTLVL or PME. 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.

(The ASTLVL and PME fields of the PCB are contained in registers when the process is executing. In order to access them, two privileged registers are provided. These are the AST Level Register and the Performance Monitor Enable Register (PME)).

ASYNCHRONOUS SYSTEM TRAPS (AST)

Asynchronous system traps are used to notify a process of events that are not synchronized with its execution and to initiate processing for such events with the least possible delay. The delay in delivery may be due to either process nonresidence or an access mode mismatch. The efficient handling of ASTs in the VAX family processors requires some hardware assistance to detect changes in access mode (current access mode in PSL). Each of the four execution access modes (kernel, executive, supervisor, and user) may receive ASTs; however, an AST for a less privileged access mode must not be permitted to interrupt execution in a more privileged access mode. Since transitions to a less privileged access mode occur only in the Return from Exception or Interrupt instruction (described in Chapter 10), 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 to put an AST control block in the queue to the software PCB; software then sets the ASTLVL field in the hardware PCB to the most privileged access mode for which an AST is pending. If the target process is currently executing, the ASTLVL privileged register also has to be set.
2. When an REI instruction detects a transition to an access mode that can be interrupted by a pending AST, an IPL 2 interrupt is requested to cause delivery of the AST. Note that the REI instruction does not make pending AST checks while returning to a routine executing on the interrupt stack.
3. The IPL 2 interrupt service routine computes the correct new val-

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

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

Note that two of the software interrupt priority levels are reserved for process structure software: IPL 2 is for AST delivery interrupts and IPL 3 is for process scheduling interrupts.

PROCESS STRUCTURE INSTRUCTIONS

Process scheduling software executes on the interrupt stack (PSL<IS> set) in order to have a non-context-switched stack available for use. If the scheduler were running on a process's kernel stack, then any state information it had there would disappear when a new process is selected. Running on the interrupt stack can occur as the result of the interrupt origin of scheduling events; however, some synchronous scheduling requests such as a WAIT service may cause rescheduling without any interrupt occurrence. For this reason, the Save Process Context (SVPCTX) instruction can be executed while on either the kernel or interrupt stack; it 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. Descriptions of them may be found in Chapter 10, Privileged and Miscellaneous Instructions.

digital
VAX 11/780

CHAPTER 9
EXCEPTIONS AND INTERRUPTS

INTRODUCTION

At certain times during system operation, events within the system may require the execution of particular "pieces" of software outside the explicit flow of control. The processor forces a change in the flow of control from that which would be explicitly indicated in the currently executing process.

Some such events are relevant to the current process and normally invoke softwarè in the context of the current process. The notification of these events is termed an **exception**.

Other events are relevant to other processes, or to the system as a whole, and are serviced in a systemwide context. The notification process for these events is termed an **interrupt**, and the systemwide 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 moment. The priority associated with an interrupt is termed its **interrupt priority level** (IPL).

EVENT HANDLING

Exceptions are handled by the operating system. Usually, they are "reflected" to the originating mode as a signal. In general, the exception is described by a vector that is a list of longwords, the first of which contains a count of other longwords in the vector. The second longword identifies which exception occurred, and the remaining longwords are the stack parameters, the PC, and the PSL, as described in this chapter. Three kinds of exceptions are explained immediately below.

A *trap* is an exception condition that occurs at the end of the instruction that caused the exception. Therefore, the PC saved on the stack is the address of the next instruction that would normally have been executed. Any software can enable and disable some of the trap conditions with a single instruction; see, for example, the descriptions of the BISPSW and BICPSW instructions.

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

An *abort* is an exception condition that occurs during an instruction, and potentially leaves the registers and memory indeterminate, such that the instruction cannot necessarily be correctly restarted, completed, simulated, or undone.

The processor arbitrates *interrupt* requests according to priority. Only when the priority of an interrupt request is higher than the current IPL (bits<20:16> of the Processor Status Longword) does the processor raise the IPL and service the interrupt request. The interrupt service routine is entered at the IPL of the interrupt request and does not usually change the IPL set by the processor.

Interrupt requests come from devices, controllers, other processors (in customer-designed systems), or the processor itself. Software that is executing in kernel mode can also raise and lower the priority of the processor. But note that the priority level of one processor does not affect the priority level of the other processors, so that interrupt priority levels cannot be used to synchronize access to shared resources in multiprocessor systems. Special software action is required to stop other processors in your multiprocessor system.

Most service routines for software-generated exceptions execute at IPL 0. However, if a serious system failure occurs, the processor raises the IPL to the highest level ($1F_{16}$) to prevent interruption until the problem is corrected. Exception service routines are usually coded to avoid exceptions; however, nested exceptions may occur (but rarely) in the case of an access control violation, reserved operand, or reserved addressing mode fault.

Processor Interrupt Priority Levels (IPLs)

The processor has 31 interrupt priority levels (IPLs), divided into 15 software levels (numbered 1_{16} to F_{16}) and 16 hardware levels (10_{16} to $1F_{16}$). User applications, system calls, and system services all run at IPL 0, which we call *process level*. Higher numbered IPLs have higher priority; that is to say, any requests at an interrupt level higher than the processor's current IPL interrupt immediately, but requests at a lower or equal level are deferred.

Interrupt levels 1 through F_{16} exist entirely for use by software. No hardware device can request interrupts on those levels, but software can force an interrupt by executing MTPR src,#SIRR (Software Interrupt Request Register). Once a software interrupt request is made, it is cleared by hardware when the interrupt is taken.

Interrupt levels 10_{16} to 17_{16} are for use by devices and controllers, including UNIBUS devices.

Interrupt levels 18_{16} to $1F_{16}$ are used by urgent conditions, including the interval clock, serious errors, and power fail.

Contrast Between Exceptions and Interrupts

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

1. An exception condition is caused by the execution of the current instruction, while an interrupt is caused by some activity in the computing system that usually is independent of the current instruction.
2. An exception condition usually is serviced in the context of the process that produced the exception condition, while an interrupt is serviced independently of the current process.
3. The IPL of the processor usually is not changed when the processor initiates an exception, while the IPL always is raised when an interrupt is serviced.
4. Exception service routines usually execute on a per-process stack, while interrupt service routines normally execute on a per-processor stack. Machine check always executes on the ISP, however.
5. Enabled exceptions are initiated immediately, independent of the processor IPL. Interrupts, however, are delayed until the processor IPL drops below the IPL of the requesting interrupt.
6. Most exceptions cannot be disabled. But, if an exception-causing event occurs while that exception is disabled, no exception is initiated for that event, even when enabled subsequently. This includes overflow, which is the only exception whose occurrence is indicated by a condition code (V). If an interrupt condition occurs while that interrupt is disabled, or the processor is at the same or higher IPL, the condition eventually initiates an interrupt when the proper enabling conditions are met (if the condition is still present).
7. The previous mode field in the PSL is always set to kernel on an interrupt, but on an exception it indicates the mode in which the exception occurred.

INTERRUPTS

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

The following events cause interrupts:

- Device completion (IPL 10-17_{16})
- Device error (IPL 10-17_{16})
- Device alert (IPL 10-17_{16})
- Device memory error (IPL 10-17_{16})
- Console terminal transmit and receive (IPL 14_{16})
- Console storage device (IPL 17_{16} for VAX-11/750 and IPL 14_{16} for VAX-11/780)
- Interval timer (IPL 18_{16})
- Recovered memory, bus or processor errors (the VAX-11/750 interrupts at IPL $1A_{16}$ for corrected memory reads; the VAX-11/780 at IPL $1B_{16}$, implementation specific)
- Unrecovered memory, bus or processor errors (the VAX-11/750 and VAX-11/780 interrupt at IPL $1D_{16}$ for write memory errors, implementation specific)
- Power fail (IPL $1E_{16}$)
- Software interrupt invoked by MTPR #SIRR (IPL 1 to F_{16})
- AST delivery when REI restores a PSL with IS clear and mode greater than or equal to ASTLVL (IPL 2_{16})

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

In order to reduce interrupt overhead, no memory mapping information is changed when an interrupt occurs. 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_{16}-$1F_{16}$

The processor has eight priority levels for use by urgent conditions, including serious errors (e.g., machine check) and power fail. 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.

Device Interrupts—Levels 10_{16}-17_{16}

The processor provides eight priority levels for use by peripheral devices. Any given implementation may or may not have all levels of interrupts. On the VAX-11/750, for example, only levels 14 through 17 are available for device interrupts.

Software-Generated Interrupts—Levels 1_{16}-F_{16}

The processor provides 15 priority interrupt levels for use by software. For details, see the VAX Software Handbook in this Handbook set.

Software Interrupt Summary Register

The Software Interrupt Summary Register (SISR) is a privileged register which records pending software interrupts. It contains 1s 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.

Software Interrupt Request Register

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

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

Interrupt Priority Level Register

Writing to the IPLR with the MTPR instruction will load the processor priority field in the Processor Status Longword (PSL). That is, bits<20:16> of the PSL are loaded from IPLR<4:0>. Reading from IPLR with the MFPR instruction will read the processor priority field from the PSL.

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

Interrupt Example

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

Table 9-1 Interrupt Sequence Example

	State After	Event	IPL In
	Contents of	SISR	PSL
On			
Event	IPL(hex)	(hex)	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
IPL 9 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
IPL 7 service routine REI	5	8	0
initial IPL 5 service routine REI back to IPL 0 and the request for 3 is granted immediately	3	0	0
IPL 3 service routine REI	0	0	—

Serious System Failures

Serious system failures are exceptions which are processed by privileged software.

Kernel stack not valid abort is an exception that indicates that the kernel stack was not valid while the processor was pushing information onto the stack during the initiation of an exception or interrupt. Usually this is an indication of stack overflow or another executive software error. The attempted exception is transformed into an abort that uses the interrupt stack. No information other than the PSL and PC is pushed onto the interrupt stack. The IPL is raised to $1F_{16}$. Software may abort the process without aborting the system; however, because of the lost information, the process cannot be continued. If the kernel stack is not valid during the normal execution of an instruction (including CHMK or REI), the processor initiates the normal memory management fault, and if the exception vector <1:0> for kernel stack not valid is 0 or 3, the behavior of the processor is undefined.

An interrupt stack not valid halt is an exception indicating that the interrupt stack was not valid or that a memory error occurred while the processor was pushing information onto the stack during the initiation of an exception or interrupt. No further interrupt requests are acknowledged on this processor. The processor leaves the PC, the PSL, and the reason for the halt in registers so that they are 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.

A machine check exception indicates that the processor detected an internal error in itself. Machine check exceptions can be caused by such bus errors as non-existent memory, cache parity, translation buffer parity, or by a control store parity error. Like other exceptions, this exception is taken independently of IPL. IPL is raised to $1F_{16}$. A length parameter, an error code, and the contents of several registers are pushed onto the stack as longwords. The processor specifies the length parameter by placing the number of bytes pushed as the last longword pushed. This count excludes the PC, PSL, and the length parameter itself. Software decides, on the basis of the information presented, whether to abort the current process if the machine check came from the process. Machine check includes uncorrected bus and memory errors, and any other processor-detected errors. Some processor errors cannot ensure the state of the machine at all. For such errors, the state will be preserved on a "best effort" basis. If the exception vector <1:0> for machine check is 0 or 3, the behavior of the processor is undefined. Under these conditions, the VAX processor will halt. Figure 9-1 shows the format of the stack after machine check exceptions, on a VAX-11/750.

Arithmetic Exceptions

The various exceptions that arise as a result of arithmetic or conversion operations are mutually exclusive and therefore can all be assigned the same vector in the System Control Block. Each indicates

LENGTH PARAMETER (28_{16})
ERROR CODE
VA REGISTER
PC AT TIME OF ERROR
MDR
SAVED MODE REGISTER
READ LOCK TIMEOUT
TB GROUP PARITY ERROR REGISTER
CACHE ERROR REGISTER
BUS ERROR REGISTER
MACHINE CLOCK ERROR SUMMARY REGISTER
PC
PSL

Figure 9-1 Machine Check Exception Stack

that an exception occurred during the last instruction and that the instruction has been either completed (in the case of a trap) or backed up (fault). A code unique to each exception type is then pushed on the stack as a longword. Figure 9-2 illustrates the stack after the occurrence of an arithmetic exception. Note that in the case of a fault, the PC of the next instruction will be the same as the instruction which caused the exception.

TYPE CODE
PC OF NEXT INSTRUCTION TO EXECUTE
PSL

Figure 9-2 Stack After Arithmetic Exception

The specific type codes saved are described in Table 9-2.

Table 9-2 Arithmetic Exception Type Codes

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

Description of Traps and Faults

An *integer overflow trap* is an exception indicating that the last instruction executed had an integer overflow which set the V condition code. This trap only occurs if the integer overflow enable bit (IV) in the PSW is set. The result stored is the low order part of the correct result. The type code pushed on the stack is a 1.

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

A *floating overflow trap* is an exception that indicates that the last instruction executed resulted in an exponent greater than 127 (unbiased) after normalization and rounding. The result is stored as a 1 in the sign and 0s in the exponent and fraction fields. This is a reserved operand, and will cause a reserved operand fault if used in a subsequent floating point instruction. The type code pushed on the stack is 3. The floating overflow trap is not implemented on VAX-11/750 systems.

A *decimal string divide by zero trap* is an exception indicating that the last instruction executed had a decimal string zero divisor. The destination and condition codes are unpredictable. The zero divisor can be either +0 or −0. The type code pushed on the stack is 4.

A *floating underflow trap* is an exception that indicates that the last instruction executed resulted in an exponent less than −127 (unbiased) after normalization and rounding and that floating underflow was enabled (FU set). The result stored is 0. The type code pushed on the stack is 5. This trap is not implemented on VAX-11/750 systems.

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

A *subscript range trap* is an exception indicating that the last instruction was an INDEX instruction with a subscript operand that failed the range check. The value of the subscript operand is lower than the low operand or greater than the high operand. The result is stored in indexout, and the condition codes are set as if the subscript were within range. The type code pushed on the stack is 7.

A *floating overflow fault* is an exception indicating that the last instruction executed resulted (after rounding and normalization) in an exponent greater that the largest representable exponent for the data type. The destination is unaffected and the saved condition codes are unpredictable. The saved PC points to the instruction causing the fault. The type code pushed on the stack is 8.

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

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

This fault does not appear on VAX-11/780 systems unless the optional G_floating and H_floating data types are implemented.

SYSTEM CONTROL BLOCK (SCB)

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

The system control block base is a privileged register containing the physical address of the System Control Block, which must be page-aligned.

Vectors

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

Separate vectors are defined for each interrupting device controller and each class of exception.

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

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

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

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

3 Operation undefined. Reserved to DIGITAL.

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

System Control Block (Exception and Interrupt Vectors)

Vector (hexadecimal)	Name	Type	Notes
00	Unused		Reserved to DIGITAL.
04	Machine Check	abort/ trap	Length parameter and

Vector (hexadecimal)	Name	Type	Notes
			error specific data are pushed onto the stack, if possible. The VAX processor is sometimes restartable, if the cause of the machine check was a cache parity, translation buffer parity, or uncorrected data error. Vector <1:0> must be 1 for meaningful operation. IPL is raised to $1F_{16}$. The number of bytes of parameters is pushed onto the stack.
08	Kernel Stack Not Valid	abort	Vector <1:0> must be 1 for meaningful operation. IPL is raised to $1F_{16}$. There are zero parameters.
0C	Power Fail	interrupt	IPL is raised to $1E_{16}$. There are zero parameters.

Vector (hexadecimal)	Name	Type	Notes
10	Reserved/ Privileged Instruction	fault	Opcodes reserved to DIGITAL and privileged instructions. There are zero parameters. On the VAX-11/750, attempting an REI with PSL<FPD> = 1 to an instruction that does not set FPD will also be reported here.
14	Customer Reserved Instruction	fault	XFC instruction. There are zero parameters.
18	Reserved Operand	fault/ abort	There are zero parameters.
1C	Reserved Addressing Mode	fault	There are zero parameters.
20	Access Control Violation	fault	The virtual address causing the fault is pushed onto the kernel stack. There are two parameters.
24	Translation Not Valid	fault	The virtual address causing the fault is

Vector (hexadecimal)	Name	Type	Notes
			pushed onto stack. The two parameters are identical to those for access control violations.
28	Trace Pending (TP)	fault	This vector is used for a trace fault. There are zero parameters.
2C	Breakpoint Instruction	fault	This vector is used for a breakpoint fault. There are zero parameters.
30	Compatibility	fault/ abort	A type code is pushed onto the stack. There is one parameter.
34	Arithmetic	trap/fault	A type code is pushed onto the stack. There is one parameter.
38-3C	Unused		Reserved to DIGITAL.
40	CHMK	trap	The operand word is sign-extended and pushed onto the kernel stack. Vector <1:0> MBZ. There is one parameter.

Vector (hexadecimal)	Name	Type	Notes
44	CHME	trap	The operand word is sign-extended and pushed onto the executive stack. Vector <1:0> MBZ. There is one parameter.
48	CHMS	trap	The operand word is sign-extended and pushed onto the supervisor stack. Vector <1:0> MBZ. There is one parameter.
4C	CHMU	trap	The operand word is sign-extended and pushed onto the user stack. Vector <1:0> MBZ. There is one parameter.
50	SBI SILO Compare	interrupt	IPL is 19_{16}. VAX-11/780 only.
54	Corrected Memory Read Data	interrupt	IPL is $1A_{16}$. Also used for Read Data Substitute on VAX-11/780.
58	SBI Alert	interrupt	IPL is $1B_{16}$. VAX-11/780 only.

Vector (hexadecimal)	Name	Type	Notes
5C	SBI Fault	interrupt	IPL is $1C_{16}$. VAX-11/780 only.
60	Memory Write Timeout	interrupt	IPL is $1D_{16}$.
64-80	Unused		Reserved to DIGITAL.
84	Software Level 1	interrupt	There are zero parameters.
88	Software Level 2	interrupt	Ordinarily used for AST delivery. There are zero parameters.
8C-BC	Software Level 3	interrupt	There are zero parameters.
90-BC	Software Levels 4-F	interrupt	There are zero parameters.
C0	Interval Timer	interrupt	IPL is 18_{16}. There are zero parameters.
C4-DC	Unused		Reserved to DIGITAL.
E0-EC	Unused		Reserved to CSS/customers.
F0	Console Storage Device (TU58) Transmit	interrupt	IPL is 17_{16}. VAX-11/750 only. There are zero parameters.
F4	Console Storage Device (TU58) Transmit	interrupt	IPL is 17_{16}. VAX-11/750 only. There are zero parameters.

Vector (hexadecimal)	Name	Type	Notes
F8	Console Terminal Receive	interrupt	IPL is 14_{16}. There are zero parameters.
FC	Console Terminal Receive	interrupt	IPL is 14_{16}. There are zero parameters.
100-3FC	Device Vectors		In the VAX-11/780 processor, only interrupt priority levels 14 to 17 are available to a NEXUS external to the CPU, and there is a limit of 16 such NEXUSes. A NEXUS is a connection on the SBI, which is the internal interconnection structure. The NEXUS vectors are assigned as follows: 100-13C IPL 14_{16} NEXUS 0-15 140-17C IPL 15_{16} NEXUS 0-15 180-1BC IPL 16_{16} NEXUS 0-15 1C0-1FC IPL 17_{16} NEXUS 0-15

Vector (hexadecimal)	Name	Type	Notes
			In the VAX-11/750 processor, UNIBUS devices interrupt the processor directly. The vector is determined by adding 200_{16} to the vector supplied by the device. Only SCB vectors in the range 200 to 3FC are allowed. Interrupt priority levels 14 to 17 correspond to UNIBUS BR4 to BR7.

STACKS

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

Stack Residency

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

If either of these faults occurs on a reference to the kernel stack, the processor aborts the current sequence and initiates a kernel stack not valid abort on hardware level $1F_{16}$. If either fault occurs on a reference to the interrupt stack, the processor halts. The kernel stack for processes other than the current one need not be resident, but it must be resident before the software's process dispatcher selects a process to run.

Stack Alignment

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

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

Stack Status Bits

The interrupt stack bit (IS) and current mode bits in the privileged 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 the current mode to be nonzero when IS = 1. This is achieved by clearing the mode bits when taking an interrupt or exception, and by causing a reserved operand fault if REI attempts to load a PSL in which both IS and mode are nonzero.

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

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

Figure 9-3 Stack Selection

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

Accessing Stack Registers

The processor implements five privileged registers to allow access to each stack pointer. These registers always access the specified pointer, even for the current mode. Because the per-process stack pointers are implemented as internal registers, the MTPR and MFPR of these registers do not access the hardware PCB. The register numbers were chosen to be the same as PSL<26:24>. The previous stack pointer is the same as PSL<23:22> unless PSL<IS> is set. Figure 9-4 illustrates the process stack pointer.

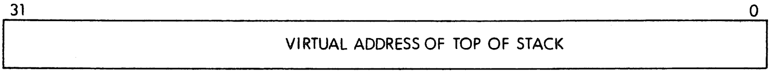


Figure 9-4 Process Stack Pointer Implemented As Read/Write Register

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 INTERRUPTS

The sequence in which recognition of simultaneously occurring interrupts and exceptions takes place is:

1. Machine check exception.
2. Arithmetic exceptions.
3. Console halt or higher priority interrupt. (The order in which console halt and interrupt recognition occur is not dictated by the VAX architecture. Some future VAX machines may not take these in the same order as the VAX-11/750 or VAX-11/780, which take console halts before interrupts.)
4. Trace fault (only one per instruction).
5. Start instruction execution or restart suspended instruction.

Suspended Instructions

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

CHAPTER 10
PRIVILEGED AND MISCELLANEOUS INSTRUCTIONS

VAX processors provide several categories of instructions that authorize greater privilege, under strictly controlled conditions, than the executing software would normally have. These are called the **privileged** instructions. Another category, the **miscellaneous** instructions, includes a variety of actions that don't fall into other classifications.

I. Change Mode, PROBE, Return from Exception or Interrupt

Since the typical user's programs run in the low-privileged user mode, it is important to allow them a way to exploit system services that run at a higher level of privilege, but not to allow them access to other modes (kernel, executive, supervisor) except for these necessary services. The privileged instructions satisfy this need. They give upward and downward mobility through the processor modes, and provide a way to compare memory protection levels against the privilege of callers. They are:

- 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 (upward) only. When such a mode transition takes place, the previous mode is saved in the Previous Mode field of the Processor Status Longword, allowing the more privileged code to determine the privilege of its caller.

A Change Mode instruction is simply a special trap instruction that can be thought of as an operating system service call instruction. User mode software can explicitly issue Change Mode instructions, but since the operating system receives the trap, nonprivileged users cannot write any code to execute in any of the privileged access modes. User mode software can include a condition handler for Change Mode to User traps, however, and this instruction is useful for providing general-purpose services for user mode software. Ultimately, though, it is the system manager who 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 so that the process resumes at the point where it was interrupted.

REI performs special services, however, that normal return instructions do not. For example, REI checks to see if any asynchronous system traps have been queued for the currently executing process while the interrupt or exception service routine was executing, and ensures that the process will receive them. Furthermore, REI checks to ensure that the mode to which it is returning control is the same as or less privileged than the mode in which the processor was executing when the exception or interrupt occurred. Thus, REI is available to all software including user-written trap handling routines, but a program cannot increase its privilege by altering the processor state to be restored.

II. Save and Load Process Context

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.

III. Move to and Move from Processor Register

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

IV. Miscellaneous Instructions

The Extended FunCtion (XFC) provides a controlled mechanism for software to request services of nonstandard microcode in the writeable control store or simulator software running in kernel mode. The request is controlled by the contents of the System Control Block. Other miscellaneous instructions include the Breakpoint Fault instruction, used to stop a process for debugging purposes, and HALT, which stops the processor.

CHM

CHANGE MODE

Purpose: request services of more privileged software

Format: opcode code.rw

Operation:

```
tmp1 ← {mode selected by opcode      (K=0, E=1, S=2, U=
3)};
tmp2 ← MINU(tmp1, PSL<CUR_MOD>);
!maximize privilege
tmp3 ← SEXT(code);
if {PSL<IS> EQLU 1} then HALT;        !illegal from I stack

PSL<CUR_MOD>_SP ← SP;                 !save old stack pointer
tmp4 ← tmp2_SP;                       !get new stack pointer
PROBEW (from tmp4−1 through tmp4−12 with mode=tmp2);
   !check       !new stack access

                if {access control violation} then
                    {initiate access violation fault};
                if {translation not valid} then
                    {initiate translation not vaid fault};

{initiate CHMx exception with new_mode=tmp2
                    and parameter=tmp3
                    using 40+tmp1*4 (hex) as SCB offset
                    using tmp4 as the new SP
                    and not storing SP again};
```

Condition Codes:

```
N ← 0;
Z ← 0;
V ← 0;
C ← 0
```

Exceptions: Halt

Opcodes:

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

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

```
sign extended code      :(SP)

     PC of next instruction

          old PSL
```

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

Notes:

1. As usual for faults, any access violation or translation not valid fault saves PC, PSL, and leaves SP as it was at the beginning of the instruction except for any pushes onto the kernel stack.
2. The noninterrupt stack pointers may be fetched and stored either in privileged registers or in their allocated slots in the PCB. Only LDPCTX and SVPCTX always fetch and store in the PCB. MFPR and MTPR always fetch and store the pointers whether in registers or the PCB.
3. By Software convention, negative codes are reserved to DIGITAL 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 service
               ;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<PRV_MOD>)
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; the bytes in between are not checked. System software must check all pages between the two end bytes if they are to be accessed.

The protection is checked against the larger of the modes specified in bits <1:0> of the mode operand and the previous 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>.

Example:

```
MOVL    4(AP),R0   ;copy address of first argument so
                   ;that it can't be changed
PROBER    #0,#4,(R0)
;verify that the longword pointed
                   ;to by the first argument could be
                   ;read by the previous access mode
                   ;Note that the argument list itself
                   ;must already have been probed
MOVQ    8(AP),R0   ;copy length and address
                   ;of buffer arguments so that
                   ;they can't change
PROBEW    $0,R0,(R1)
;verify that the buffer described
                   ;by the second and third arguments
                   ;could be written by the previous
                   ;access mode
                   ;Note that the argument list must
                   ;already have been probed and that
                   ;the second argument must be known
                   ;to be less than 512
```

REI

RETURN FROM EXCEPTION OR INTERRUPT

Purpose: exit from an exception or interrupt service routine and controlled return

Format: Opcode

Operation:

```
tmp1 ← (SP) +;   !Pick up saved PC
tmp2 ← (SP) +;   !and PSL

if  {tmp2<current_mode> LSSU PSL<current_mode>} OR
    {tmp2<IS> EQLU 1 and PSL<IS>EQLU 0} OR
    {tmp2<IS> EQLU 1 and
    tmp2<current_mode> NEQU 0} OR
    {tmp2<IS> EQLU 1 and tmp2<IPL> EQLU 0} OR
    {tmp2<IPL> GRTU 0 and
    tmp2<current_mode> NEQU 0} OR
    {tmp2<previous_mode> LSSU
    tmp2<current_mode>} OR
    {tmp2<IPL> GTRU PSL<IPL>} OR
    {tmp2<PSL_MBZ> NEQU 0} then
    {reserved operand fault};
if  {tmp2<CM> EQLU 1} and
    {tmp2<FPD,IS,DV,FU,IV> NEQU 0} OR
    {tmp2<current_mode> NEQU 3}} then {reserved
    operand fault};
if PSL<IS> EQLU 1 then ISP ← SP
                                          !save old stack pointer
    else PSL<current_mode>_SP ← SP;
if PSL<TP> EQLU 1 then tmp2<TP> ← 1;
                                          !TP←TP or stack TP
PC ← tmp1;
PSL ← tmp2;
if PSL<IS> EQLU 0 then
    begin
    SP ← PSL <current_mode>_SP;   !switch stack
    if PSL<current_mode> GEQU ASTLVL
                                          !check for AST
                                          delivery
    then {request interrupt at IPL 2};
end;
{check for software interrupts}:
```

Condition Codes:

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

Exceptions: Reserved operand

Opcodes: 02 REI Return from Exception or Interrupt

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

Notes:

1. The exception or interrupt service routine is responsible for restoring any registers saved and removing any parameters from the stack.
2. As usual for faults, any access violation or translation not valid conditions on the stack pops restore the stack pointer and fault.
3. The noninterrupt stack pointers may be fetched and stored by hardware either in internal registers or in their allocated slots in the Process Control Block. Only LDPCTX and SVPCTX always fetch and store in the Process Control Block. MFPR and MTPR always fetch and store the pointers whether in registers or the Process Control Block.

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};
   !LDPCTX}
{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
Privileged 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.

Example: It is assumed that this simple dispatch routine 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
```

MFPR
MTPR

MOVE FROM PROCESSOR REGISTER
MOVE TO PROCESSOR REGISTER

Purpose: provide access to the internal privileged registers

Format:

```
opcode src.rl, procreg.rl                         !MTPR
opcode procreg.rl, dst.wl                         !MFPR
```

Operation:

```
if PSL<current-mode> NEQ 0 then
{reserved instruction fault};
PRS [procreg] ←src;                               !MTPR
dst   ←PRS[procreg];                              !MFPR
```

Condition Codes:

```
N ←dst LSS 0;   !if register/destination is replaced
Z ←dst EQL 0;
V ←0;
C ←C

N ←N;   !if register/definition is not replaced
Z ←Z;
V ←V;
C ←C
```

Exceptions: Reserved operand
Privileged instruction

Opcodes:

DA	MTPR	Move to Processor Register
DB	MFPR	Move from Processor Register

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

Notes:

1. A reserved operand fault may occur if the processor internal register does not exist.
2. A reserved instruction fault occurs if instruction execution is attempted in other than kernel mode.

Example: The following table is a summary of the registers accessible in the privileged register space.

The "type" column indicates read-only (R), read/write (R/W), or write-only (W) characteristics.

"Scope" indicates whether a register is per-CPU or per-process. The implication is that, in general, registers labeled "CPU" are manipulated only through software via the MTPR and

MFPR instructions. Per-process registers, on the other hand, are manipulated implicitly by context switch instructions. The "Init" column indicates that the register is ("yes") or is not ("no") set to some predefined value (note: not necessarily cleared) by a processor initialization command. A "—" indicates initialization is optional.

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

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

VAX Series Registers

Register Name	Mnemonic	Number$_{16}$	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	PROC	—
P1 Base Register	P1BR	10	R/W	PROC	—
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	—

Register Name	Mne- monic	Num- ber$_{16}$	Type	Scope	Init
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

XFC

EXTENDED FUNCTION CALL

Purpose: provides customer-defined 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 nonstandard microcode or software. If no special microcode is loaded, then an exception is generated to a kernel mode software simulator (see Chapter 3). Typically, the next byte would specify which of several extended functions are requested. Parameters would be passed either as normal operands, or, more likely, in fixed registers.

BPT

BPT BREAKPOINT FAULT

Purpose: helps implement debugging

Format: opcode

Operation:

```
PSL<TP> ← 0;
{breakpoint fault};    !Push current PSL on stack
```

Condition Codes:

```
N ← 0;    !condition codes cleared after BPT fault
Z ← 0;
V ← 0;
C ← 0
```

Exceptions: None

Opcodes: 03   BPT   Breakpoint Fault

Description: This instruction is used, together with the T-bit, to implement debugging facilities.

BUGCHECK

Purpose:

Format: opcode message.ix

Operation: {fault to report error}

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

Exceptions: Reserved instruction

Opcodes:

FEFF	BUGW	Bugcheck with word message identifier
FDFF	BUGL	Bugcheck with longword message identifier

Description: The hardware treats these opcodes as RESERVED to DIGITAL and faults. The VAX/VMS operating system treats these as requests to report software-detected errors. The in-line message identifier is zero-extended to a longword (BUGW) and interpreted as a condition value. If the process is privileged to report bugs, a log entry is made. If the process is not privileged, a reserved instruction is signalled.

HALT

HALT

Purpose: stop processor operation

Format: opcode

Operation:

```
If PSL<current_mode> NEQU kernel then
   {privileged instruction fault}
   else
   {halt the processor};
```

Condition Codes:

```
N ← 0;  !If privileged instruction fault
Z ← 0;  !condition codes are
V ← 0;  !cleared after the fault.
C ← 0

N ← N;  !If processor halt
Z ← Z;
V ← V;
C ← C
```

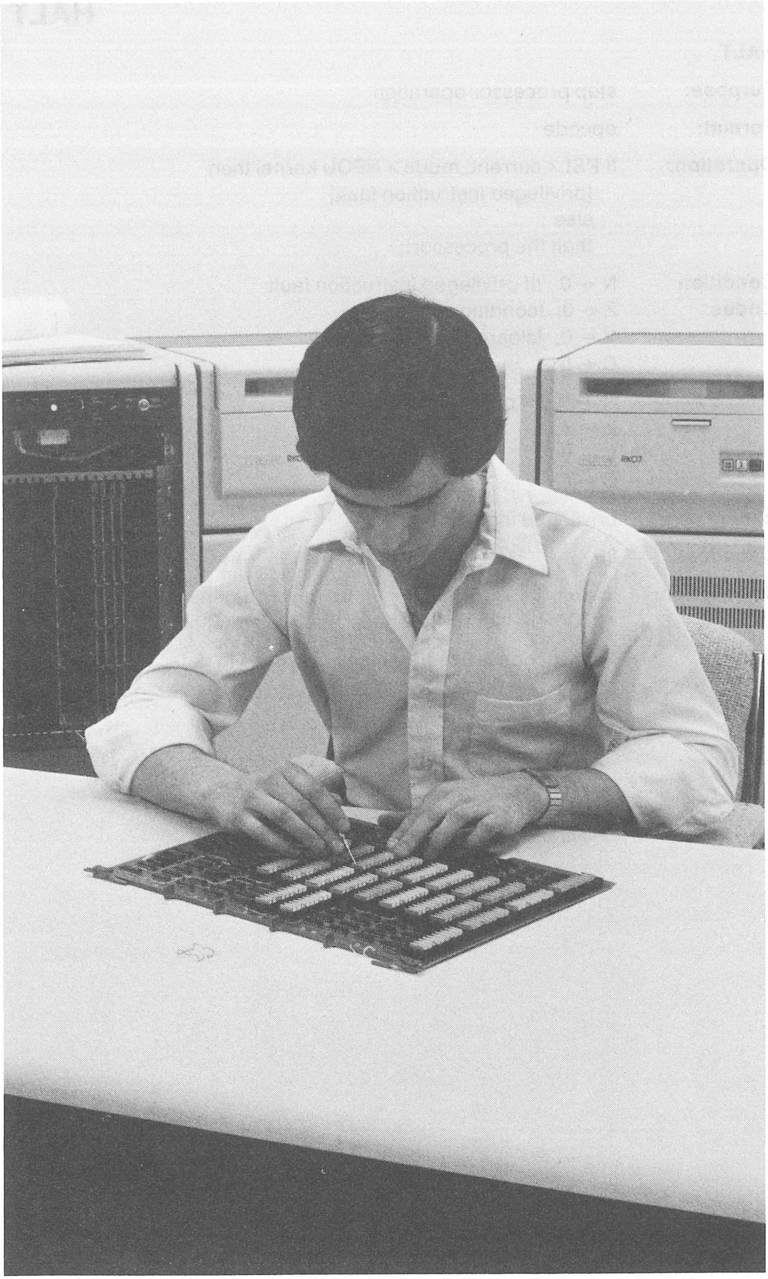
Exceptions: privileged instructions

Opcodes:

00	HALT	Halt

Description: If the process is running in kernel mode, the processor is halted. Otherwise, a privileged instruction fault occurs.

Notes: This opcode is 0 to trap many branches to data.



CHAPTER 11
INTEGER AND FLOATING POINT INSTRUCTIONS

INSTRUCTION SET OVERVIEW

A major goal of the VAX architecture is to provide an instruction set that is symmetric with respect to data types. For example, there are separate ADD instructions for seven integer and floating point data types (byte, word, longword, F_, D_, G_, and H_floating), each available in both two-operand and three-operand format (e.g., ADDB2 and ADDH2). Other symmetric operations include data movement, data conversion, data testing, and computation. Thus, both assembly language programmers and compilers can choose the best instruction to use independent of the data type.

To simplify understanding of the instruction set, the instruction mnemonics are formed by combining an operation prefix with a data type suffix. Convert instructions, for example, are formed by adding suffixes for both the source and destination data types, as in CVTGH, convert G_floating to H_floating. The computation instructions include a further suffix to indicate the choice between two-operand and three-operand instructions. And special instruction mnemonics have been chosen for similarity. Table 11-1 shows you some instruction mnemonics. For example, a Move Word instruction has the mnemonic MOVW, while a Move F_floating instruction has the mnemonic MOVF.

Table 11-1 Integer, Floating Point, and Optimization Instructions

Instruction	**Data Type**	**Number of Operands**
MOVe	Byte	
CLearR	Word	
	Longword	
	Quadword	1 operand
	Octaword	
	F_floating	
	D_floating	
	G_floating	
	H_floating	

Instruction	Data Type		Number of Operands
Move NEGated CoMPare TeST	Byte Word Longword F_floating D_floating G_floating H_floating		1 operand
Move COMplemented BIt Test	Byte Word Longword		
ConVerT	Byte Word Longword F_floating D_floating G_floating H_floating	Byte Word Longword F_floating D_floating G_floating H_floating	
	except BB, WW, LL, FF, DD, GG, HH, DG, and GD		
ADD SUBtract MULtiply DIVide	Byte Word Longword F_floating D_floating G_floating H_floating		2 operand 3 operand
BIt Set BIt Clear eXclusive OR	Byte Word Longword		2 operand 3 operand
Extended MODulus POLYnominal evaluation	F_floating D_floating G_floating H_floating		
PUSH Longword	Longword		

Instruction	Data Type	Number of Operands
INCrement DECrement	Byte Word Longword	
MOVe Zero-extended	Byte to Word Byte to Longword Word to Longword	
ConVerT Rounded	F_Floating to Longword D_floating to Longword G_floating to Longword H_floating to Longword	
ADD Aligned Word under memory Interlock ADd With Carry SuBtract With Carry Extend MULtiply Extend DIVide		
Arithmetic Shift	Longword Quadword	
ROTate Longword	Longword	

The move operations are simple move, clear, arithmetic negate, and logical complement. The logical complement operations are available only for the three integer data types because these are the logical types. Both negate and complement include a move, rather than being restricted to altering an operand in place. VAX has a large set of converts, covering almost all data type pairs. In addition, special converts exist to round floating data to integer, and to extend unsigned integers to larger integers. The data comparison and testing instructions are comparison, test against zero, and multiple bit testing.

Computation instructions are add, subtract, multiply, and divide. The logic computation instructions are for the three integer data types and are bit set (inclusive OR), bit clear (complement AND), and exclusive OR. Arithmetic and logical computation instructions are available in both two- and three-operand forms for each applicable data type. The three-operand form takes as input the values of the first two operands and stores the result in the third operand.

The integer optimizations include an instruction to push a longword onto the stack. Each integer data type includes operations for increment and decrement. VAX includes special instructions to implement multiple precision integer arithmetic add, subtract, multiply, and divide. A special variant of integer add is an operation that adds a word under a memory interlock (for operating system counters in a multiprocessor system). VAX includes special floating point instructions for modulus (range reduction) and polynomial calculation to aid in the implementation of mathematical functions, along with shift and rotate instructions.

FLOATING POINT INSTRUCTIONS

Mathematically, a floating point number may be defined as having the form

$$\pm(2^{K})f$$

where K is an integer and f is a nonnegative fraction. For a nonvanishing number, K and f are uniquely determined by imposing the condition

$$½ \leq f < 1$$

The fraction factor, f, of the number is then said to be binary normalized. For the number 0, f must be assigned the value 0, and the value of K is indeterminate.

The VAX floating point data formats are derived from this mathematical representation for floating point numbers. Four types of floating point data are provided; F_floating numbers are 32 bits long, D_ and G_floating are 64 bits long, and H_floating numbers are 128 bits long. Chapters 2 and 4 describe the floating point formats in detail. For a refresher, you can check there, or see the appropriate data type in the Glossary.

Floating Point Zero

Because of the hidden bit, the fractional factor is not available to distinguish between zero and nonzero numbers whose fractional factor is exactly ½. Therefore VAX reserves a sign-exponent field of 0 for this purpose. Any positive floating point number with biased exponent of 0 is treated as if it were an exact 0 by the floating point instruction set. In particular, a floating point operand, whose bits are all 0s, is treated as 0, and this is the format generated by all floating point instructions for which the result is 0.

Reserved Operands

A reserved operand is defined to be any bit pattern with a sign bit of 1 and a biased exponent of 0. On VAX, all floating point instructions generate a fault if a reserved operand is encountered. Since a reserved operand has a biased exponent of 0, it can be (internally) generated only if overflow occurs.

Accuracy

An instruction is defined to be exact if its result, extended on the right by an infinite sequence of 0s, is identical to that of an infinite-precision calculation involving the same operands. The prior accuracy of the operands is thus ignored. For all arithmetic operations, except DIV, a 0 operand implies that the instruction is exact. The same statement holds for DIV if the 0 operand is the dividend. But if it is the divisor, division is undefined and the instruction traps.

We show in Appendix H that the ADD, SUB, MUL and DIV, an overflow bit on the left, and two guard bits on the right are necessary and sufficient to guarantee return of a rounded result identical to the corresponding infinite-precision operation rounded to the specified word length. Thus, with two guard bits, a rounded result has an error bound of (½) LSB (least significant bit).

Note that an arithmetic result is exact if only 0 bits are lost in chopping the infinite-precision result to the data length to be stored. The first bit lost in chopping is referred to as the "rounding" bit. The value of a rounded result is related to the chopped result as follows:

1. If the rounding bit is 1, the rounded result is the chopped result incremented by an LSB (least significant bit).
2. If the rounding bit is 0, the rounded and chopped results are identical.

Rounding may be implemented by adding a 1 to the rounding bit, and propagating the carry, if it occurs. Note that a renormalization may be required after rounding takes place; if this happens, the new rounding bit will be zero, so it can happen only once. To summarize the relations among chopped, rounded, and true (infinite-precision) results:

1. If a stored result is exact
   rounded value = chopped value = true value.
2. If a stored result is not exact, its magnitude is
   a) always less than that of the true result for chopping; and,
   b) always less than that of the true result for rounding if the rounding bit is 0; or,
   c) greater than that of the true result for rounding if the rounding bit is 1.

To be consistent with the floating point instruction set which faults on reserved operands, software-implemented floating point functions should verify that the input operands are not reserved. An easy way to do this is a move or test of the input operands.

In order to facilitate high-speed implementations of the floating point instruction set, certain restrictions are placed on the addressing mode combinations usable within a single floating point instruction. These combinations involve the logically inconsistent use of a value as both a floating point operand and an address.

Specifically: if within the same instruction the contents of Rn are used as an F_floating point operand or part of a larger floating input operand and as an address in an addressing mode which modifies Rn (i.e., autoincrement, autodecrement, or autoincrement deferred), the value of the floating point operand is unpredictable.

Appendix E gives a detailed roster of the symbols used in describing these instructions. Please refer to it for assistance.

MOV

MOVE

Purpose: move a scalar quantity

Format: opcode src.rx, dst.wx

Operation: dst ← src;

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

Exceptions: None (integer); Reserved operand (floating point)

Opcodes:

90	MOVB	Move Byte
B0	MOVW	Move Word
D0	MOVL	Move Longword
7D	MOVQ	Move Quadword
7DFD	MOVO	Move Octaword
50	MOVF	Move F_floating
70	MOVD	Move D_floating
50FD	MOVG	Move G_floating
70FD	MOVH	Move H_floating

Description: The destination operand is replaced by the source operand. The source operand is unaffected.

Notes:

1. On a floating reserved operand fault, the destination operand is unaffected and the condition codes are unpredictable.
2. Unlike the PDP-11, but like the other VAX-11 instructions, MOVB and MOVW do not modify the high order bytes of a register destination. Refer to the MOVZxL and CVTxL instructions to update the full register contents.

PUSHL

PUSH LONGWORD

Purpose: push source operand onto stack

Format: opcode src.rl

Operation: −(SP) ← src;

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

Exceptions: None

Opcodes:

DD	PUSHL	Push Longword

Description: The longword source operand is pushed onto the stack.

Notes: PUSHL is equivalent to MOVL src, −(SP), but it is one byte shorter.

CLR

CLEAR

Purpose: clear a scalar quantity

Format: opcode dst.wx

Operation: dst ← 0;

Condition Codes:

N ← 0:
Z ← 1;
V ← 0;
C ← C

Exceptions: None

Opcodes:

94	CLRB	Clear Byte
B4	CLRW	Clear Word
D4	CLRL	Clear Longword
	CLRF	Clear F_floating
7C	CLRQ	Clear Quadword
	CLRD	Clear D_floating
	CLRG	Clear G_floating
7CFD	CLRO	Clear Octaword
	CLRH	Clear H_floating

Description: The destination operand is replaced by 0.

Notes: CLRx dst is equivalent to MOVx #0,dst, but is shorter by from 1 to 17 bytes, depending on data type.

MNEG

MOVE NEGATED

Purpose: move the arithmetic negation of a scalar quantity

Format: opcode src.rx, dst.wx

Operation: dst ← −src;

Condition Codes:
N ← dst LSS 0;
Z ← dst EQL 0;
V ← {overflow} (integer);
V ← 0 (floating);
C ← dst NEQ 0 (integer);
C ← 0 (floating)

Exceptions: Integer overflow; reserved operand (floating)

Opcodes:

8E	MNEGB	Move Negated Byte
AE	MNEGW	Move Negated Word
CE	MNEGL	Move Negated Longword
52	MNEGF	Move Negated F_floating
72	MNEGD	Move Negated D_floating
52FD	MNEGG	Move Negated G_floating
72FD	MNEGH	Move Negated H_floating

Description: The destination operand is replaced by the negative of the source operand.

Notes:

1. Integer overflow occurs if the source operand is the largest negative integer (which has no positive counterpart). On overflow, the destination operand is replaced by the source operand.
2. On floating reserved operand fault, the destination operand is unaffected and the condition codes are unpredictable.

Example:

```
MOVE NEGATED FLOATING
MNEGF R0, R7             ;Replace R7 with negative
                         ;of contents of R0
```

Initial Conditions:
R0 = 00004410
R7 = 00000000

After Instruction Execution:
R0 = 00004410
R7 = 0000C410 (Change Sign Bit)

NOTE

If source is positive zero, result is positive zero. If source is reserved operand (minus zero), a reserved operand fault occurs. For all other floating point source values, bit 15 (sign bit) is complemented.

MCOM

MOVE COMPLEMENTED

Purpose: move the logical complement of an integer

Format: opcode src.rx, dst.wx

Operation: dst ← NOT src;

Condition Codes:

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

Exceptions: None

Opcodes:

92	MCOMB	Move Complemented Byte
B2	MCOMW	Move Complemented Word
D2	MCOML	Move Complemented Longword

Description: The destination operand is replaced by the one's complement of the source operand.

CVT

CONVERT

Purpose: convert a signed quantity to a different signed data type

Format: opcode src.rx, dst.wy

Operation: dst ← conversion of src;

Condition Codes:
N ← dst LSS 0;
Z ← dst EQL 0;
V ← {src cannot be represented in dst} (floating);
V ← {integer overflow};
C ← 0

Exceptions:
Integer overflow
Floating overflow
Floating underflow
Reserved operand

Opcodes:

99	CVTBW	Convert Byte to Word
98	CVTBL	Convert Byte to Longword
33	CVTWB	Convert Word to Byte
32	CVTWL	Convert Word to Longword
F6	CVTLB	Convert Longword to Byte
F7	CVTLW	Convert Longword to Word
4C	CVTBF	Convert Byte to F_floating
6C	CVTBD	Convert Byte to D_floating
4CFD	CVTBG	Convert Byte to G_floating
6CFD	CVTBH	Convert Byte to H_floating
4D	CVTWF	Convert Word to F_floating
6D	CVTWD	Convert Word to D_floating
4DFD	CVTWG	Convert Word to G_floating
6DFD	CVTWH	Convert Word to H_floating
4E	CVTLF	Convert Longword to F_floating
6E	CVTLD	Convert Longword to D_floating
4EFD	CVTLG	Convert Longword to G_floating
6EFD	CVTLH	Convert Longword to H_floating
48	CVTFB	Convert F_floating to Byte
68	CVTDB	Convert D_floating to Byte
48FD	CVTGB	Convert G_floating to Byte
68FD	CVTHB	Convert H_floating to Byte
49	CVTFW	Convert F_floating to Word
69	CVTDW	Convert D_floating to Word
49FD	CVTGW	Convert G_floating to Word
69FD	CVTHW	Convert H_floating to Word
4A	CVTFL	Convert F_floating to Longword
4B	CVTRFL	Convert Rounded F_floating to Longword

6A	CVTDL	Convert D_floating to Longword
6B	CVTRDL	Convert Rounded D_floating to Longword
4AFD	CVTGL	Convert G_floating to Longword
48FD	CVTRGL	Convert Rounded G_floating to Longword
6AFD	CVTHL	Convert H_floating to Longword
6BFD	CVTRHL	Convert Rounded H_floating to Longword
56	CVTFD	Convert F_floating to D_floating
99FD	CVTFG	Convert F_floating to G_floating
98FD	CVTFH	Convert F_floating to H_floating
76	CVTDF	Convert D_floating to F_floating
32FD	CVTDH	Convert D_floating to H_floating
33FD	CVTGF	Convert G_floating to F_floating
56FD	CVTGH	Convert G_floating to H_floating
F6FD	CVTHF	Convert H_floating to F_floating
F7FD	CVTHD	Convert H_floating to D_floating
76FD	CVTHG	Convert H_floating to G_floating

Description: The source operand is converted to the data type of the destination operand and the destination operand is replaced by the result. For integer format, conversion of a shorter data type to a longer is done by sign extension; conversion of longer to a shorter is done by truncation of the higher numbered (most significant) bits. For floating format, the form of the conversion is as follows:

CVTBF	exact	CVTHW	truncated
CVTBD	exact	CVTFL	truncated
CVTBG	exact	CVTRFL	rounded
CVTBH	exact	CVTDL	truncated
CVTWF	exact	CVTRDL	rounded
CVTWD	exact	CVTGL	truncated
CVTWG	exact	CVTRGL	rounded
CVTWH	exact	CVTHL	truncated
CVTLF	rounded	CVTHRL	rounded
CVTLD	exact	CVTFD	exact
CVTLG	exact	CVTFG	exact
CVTLH	exact	CVTFH	exact
CVTFB	truncated	CVTDF	rounded
CVTDB	truncated	CVTDH	exact
CVTGB	truncated	CVTGF	rounded
CVTHB	truncated	CVTGH	exact
CVTFW	truncated	CVTHF	rounded
CVTDW	truncated	CVTHD	rounded
CVTGW	truncated	CVTHG	rounded

Notes:

1. Integer overflow occurs if any truncated bits of the source operand are not equal to the sign bit of the destination operand.
2. Only converts with an integer destination operand can result in integer overflow. On integer overflow, the destination operand is replaced by the low-order bits of the true results.
3. Only CVTDF, CVTGF, CVTHF, CVTHD, and CVTHG can result in floating overflow. The sequence of events following floating overflow varies among the processors in the VAX family.
4. Only converts with a floating point source operand can result in a reserved operand fault. On a reserved operand fault, the destination operand in unaffected and the condition codes are unpredictable.
5. Only CVTGF, CVTHF, CVTHD, and CVTHG can result in floating underflow. As in Note 3, above, the subsequent sequence of events is processor-dependent.

Example:

CONVERT FLOATING TO WORD

```
CVTFW WORK, R0          ;Convert contents of WORK
                        ;floating to word
                        ;store in R0
```

Initial Conditions:
WORK = 00004410 (floating point 144.)
R0 = 00000000

After Instruction Execution:
WORK = 00004410
R0 = 00000090 (integer 144)

Example:

CONVERT ROUNDED FLOATING TO LONG

```
CVTRFL R2,R3        ;Converts contents of R2
                    ;floating to long, rounding
                    ;store in R3
```

Initial Conditions:
R2 = 00004332 (floating point 44.5)
R3 = 00000000

After Instructon Execution:
R2 = 00004332
R3 = 0000002D (integer 45; note the rounding)

MOVZ

MOVE ZERO-EXTENDED

Purpose: convert an unsigned integer to a wider unsigned integer

Format: opcode src.rx, dst.wy

Operation: dst ← ZEXT (src);

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

Exceptions: None

Opcodes:

9B	MOVZBW	Move Zero-Extended Byte to Word
9A	MOVZBL	Move Zero-Extended Byte to Longword
3C	MOVZWL	Move Zero-Extended Word to Longword

Description: For MOVZBW, bits 7:0 of the destination operand are replaced by the source operand; bits 15:8 are replaced by zero. For MOVZBL, bits 7:0 of the destinaton operand are replaced by the source operand; bits 31:8 are replaced by 0. For MOVZWL, bits 15:0 of the destination operand are replaced by the source operand; bits 31:16 are replaced by 0.

CMP

COMPARE

Purpose: arithmetic comparison between two scalar quantities

Format: opcode src1.rx, src2.rx

Operation: src1 – src2;

Condition Codes:
N ← src1 LSS src2;
Z ← src1 EQL src2;
V ← 0;
C ← src1 LSSU src2 (integer);
C ← 0 (floating)

Exceptions: None (integer); reserved operand (floating point)

Opcodes:

91	CMPB	Compare Byte
B1	CMPW	Compare Word
D1	CMPL	Compare Longword
51	CMPF	Compare F_floating
71	CMPD	Compare D_floating
51FD	CMPG	Compare G_floating
71FD	CMPH	Compare H_floating

Description: The source 1 operand is compared with the source 2 operand. The only action is to affect the condition codes.

Notes: On a floating reserved operand fault, the condition codes are unpredictable.

Example:

```
        CLRL      R0
FOO:      .
          .
          .
          .
          .
        INCL      R0
        CMPL      R0, 5
        BLSS      FOO
          .
          .
          .
```

The example clears the longword at R0 and proceeds through the routine at FOO. It increments the longword at R0, computes it to the number 5, and branches back to FOO on R0 less than 5.

INC

INCREMENT

Purpose: add 1 to an integer

Format: opcode sum.mx

Operation: sum ← sum + 1;

Condition Codes:
N ← sum LSS 0;
Z ← sum EQL 0;
V ← {integer overflow};
C ← {carry from most significant bit}

Exceptions: Integer overflow

Opcodes:

96	INCB	Increment Byte
B6	INCW	Increment Word
D6	INCL	Increment Longword

Description: One is added to the sum operand and the sum operand is replaced by the result.

Notes:

1. Arithmetic overflow occurs if the largest positive integer is incremented. On overflow, the sum operand is replaced by the largest negative integer.
2. INCx sum is equivalent to ADDx s↑#1, sum, but is shorter.

TST

TEST

Purpose: arithmetic compare of a scalar to 0.

Format: opcode src.rx

Operation: src – 0;

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

Exceptions: None (integer); reserved operand (floating point)

Opcodes:

95	TSTB	Test Byte
B5	TSTW	Test Word
D5	TSTL	Test Longword
53	TSTF	Test F_floating
73	TSTD	Test D_floating
53FD	TSTG	Test G_floating
73FD	TSTH	Test H_floating

Description: The condition codes are affected according to the value of the source operand.

Notes:

1. TSTx src is equivalent for floating point instructions to CMPx src, #0, but is shorter. Similarly, with CMPx src, s↑# 0, for integer instructions.
2. On a floating reserved operand, the condition codes are unpredictable.

ADD

ADD

Purpose: perform arithmetic addition

Format:

```
opcode add.rx, sum.mx                 2 operand
opcode add1.rx, add2.rx, sum.wx       3 operand
```

Operation:

```
sum ← sum + add;                      !2 operand
sum ← add1 + add2;                    !3 operand
```

Condition Codes:

N ← sum LSS 0;
Z ← sum EQL 0;
V ← overflow;
C ← carry from most significant bit (integer);
C ← 0 (floating);

Exceptions:

Integer overflow
Floating overflow
Floating underflow
Reserved operand

Opcodes:

80	ADDB2	Add Byte 2 Operand
81	ADDB3	Add Byte 3 Operand
A0	ADDW2	Add Word 2 Operand
A1	ADDW3	Add Word 3 Operand
C0	ADDL2	Add Longword 2 Operand
C1	ADDL3	Add Longword 3 Operand
40	ADDF2	Add F_floating 2 Operand
41	ADDF3	Add F_floating 3 Operand
60	ADDD2	Add D_floating 2 Operand
61	ADDD3	Add D_floating 3 Operand
40FD	ADDG2	Add G_floating 2 Operand
41FD	ADDG3	Add G_floating 3 Operand
60FD	ADDH2	Add H_floating 2 Operand
61FD	ADDH3	Add H_floating 3 Operand

Description: In 2-operand format, the addend operand is added to the sum operand and the sum operand is replaced by the result. In 3-operand format, the addend 1 operand is added to the addend 2 operand and the sum operand is replaced by the result. In floating point format, the result is rounded.

Notes:

1. Integer overflow occurs if the input operands to the add have the same sign and the result has the opposite sign. On overflow, the sum operand is replaced by the low order bits of the true result.
2. On a floating reserved operand fault, the sum operand is unaffected and the condition codes are unpredictable.
3. On floating underflow and floating overflow. The action is dependent upon the particular VAX family processor being used.

Example:

ADD FLOATING 2 OPERAND

```
ADDF2 #144., WORK                 ;ADD 144 floating point
                                  ;format to WORK
```

Initial Conditions:
WORK = 00000000

After Instruction Execution:
WORK = 00004410

Example:

ADD FLOATING 3 OPERAND

```
                  ADDF3 #144., WORK, WORK1
;Add 144 Floating
                  ;point format to contents
                  ;of WORK; store result
                  ;in WORK1
```

Initial Conditions:
WORK = 00004410 (hex); (144 floating)
WORK1 = 00000000

After Instruction Execution:
WORK = 00004410

WORK1 = 00004490 (hex); (288 floating)

ADWC

ADD WITH CARRY

Purpose: perform extended-precision addition

Format: opcode add.rl, sum.ml

Operation: sum ← sum + add + C

Condition Codes:
N ← sum LSS 0;
Z ← sum EQL 0;
V ← {integer overlow};
C ← {carry from most significant bit}

Exceptions: Integer overflow

Opcodes: D8 ADWC Add with Carry

Description: The contents of the condition code C bit and the addend operand are added to the sum operand and the sum operand is replaced by the result.

Notes:

1. On overflow, the sum operand is replaced by the low order bits of the results.
2. The two additions in the operation are performed simultaneously.

Example: ADD WITH CARRY

To add two quadword integers:

```
ADDL A, B                    ;add low half
ADWC A+4, B+4                ;add high half
                             ;including carry
```

Additional ADWC can be appended for greater precision.

ADAWI

ADD ALIGNED WORD INTERLOCKED

Purpose: maintain operating system resource usage counts

Format: opcode add.rw, sum.mw

Operation:

```
tmp ← add;
{set interlock};
sum ← sum + tmp;
{released interlock}
```

Condition Codes:

```
N ← sum LSS 0;
Z ← sum EQL·0;
V ← {integer overflow};
C ← {carry from most significant bit}
```

Exceptions: Reserved operand fault
Integer overflow

Opcodes: 58 ADAWI Add Aligned Word Interlocked

Description: The addend operand is added to the sum operand and the sum operand is replaced by the result. The operation is interlocked against similar operations by other processors or in a multiprocecessor system. The destination must be aligned on a word boundary, otherwise a reserved operand fault is taken.

Notes:

1. Integer overflow occurs if the input operands to the add have the same sign and the result has the opposite sign. On overflow, the sum operand is replaced by the low order bits of the true result.
2. If the addend and the sum operand overlap, the result and the condition codes are unpredictable.

SUB

SUBTRACT

Purpose: perform arithmetic subtraction

Format:

opcode sub.rx, dif.mx 2 operand
opcode sub.rx, min.rx, dif.wx 3 operand

Operation:

```
dif ←dif – sub;          !2 operand
dif ←min – sub;          !3 operand
```

Condition Codes:

N ← dif LSS 0;
Z ← dif EQL 0;
V ← overflow;
C ← borrow from most significant
bit (integer);
C ← 0 (floating)

Exceptions:

Integer overflow
Floating overflow
Floating underflow
Reserved operand

Opcodes:

82	SUBB2	Subtract Byte 2 Operand
83	SUBB3	Subtract Byte 3 Operand
A2	SUBW2	Subtract Word 2 Operand
A3	SUBW3	Subtract Word 3 Operand
C2	SUBL2	Subtract Longword 2 Operand
C3	SUBL3	Subtract Longword 3 Operand
42	SUBF2	Subtract F_floating 2 Operand
43	SUBF3	Subtract F_floating 3 Operand
62	SUBD2	Subtract D_floating 2 Operand
63	SUBD3	Subtract D_floating 3 Operand
42FD	SUBG2	Subtract G_floating 2 Operand
43FD	SUBG3	Subtract G_floating 3 Operand
62FD	SUBH2	Subtract H_floating 2 Operand
63FD	SUBH3	Subtract H_floating 3 Operand

Description: In 2-operand format, the subtrahend operand is subtracted from the difference operand and the difference operand is replaced by the result. In 3-operand format, the subtrahend operand is subtracted from the minuend operand and the difference operand is replaced by the result. In floating format, the result is rounded.

Notes:

1. Integer overflow occurs if the input operands to the subtract are of different signs and the sign of the result is the sign of the subtrahend. On overflow, the difference operand is replaced by the low order bits of the true result.
2. On a floating reserved operand fault, the difference operand is unaffected and the condition codes are unpredictable.

3. On floating underflow and floating overflow, the subsequent action depends on the VAX family processor being used.

Example:

```
SUBTRACT FLOATING 2 OPERAND
SUBF2 #100, WORK              ;Subtract 100 floating point
                              ;format from contents of
                              ;location WORK
```

Initial Conditions:
WORK = 00004410

After Instruction Execution:
WORK = 00004330

DEC

DECREMENT

Purpose: subtract 1 from an integer

Format: opcode dif.mx

Operation: dif ← dif – 1;

Condition Codes:
N ← dif LSS 0;
Z ← dif EQL 0;
V ← {integer overflow};
C ← {borrow from most significant bit}

Exceptions: Integer overflow

Opcodes:

97	DECB	Decrement Byte
B7	DECW	Decrement Word
D7	DECL	Decrement Longword

Description: One is subtracted from the difference operand and the difference operand is replaced by the result.

Notes:

1. Integer overflow occurs if the largest negative integer is decremented. On overflow, the difference operand is replaced by the largest positive integer.
2. DECx dif is equivalent to SUBx S↑#1, dif, but is shorter.

SBWC

SUBTRACT WITH CARRY

Purpose: perform extended-precision subtraction

Format: opcode sub.rl, dif.ml

Operation: dif ← dif − sub − C

Condition Codes:
N ← dif LSS 0;
Z ← dif EQL 0;
V ← {integer overflow};
C ← {borrow from most significant bit}

Exceptions: Integer overflow

Opcodes:

D9	SBWC	Subtract with Carry

Description: The subtrahend operand and the contents of the condition code C bit are subtracted from the difference operand and the difference operand is replaced by the result.

Notes:

1. On overflow, the difference operand is replaced by the low order bits of the true result.
2. The two subtractions in the operation are performed simultaneously.

Example: SUBTRACT WITH CARRY

To subtract two quadword integers:

```
SUBL A, B            ;subtract low half
SBWC A+4, B+4        ;subtract high half
                     ;including borrow
```

Additional SBWC can be appended for greater precision.

MUL

MULTIPLY

Purpose: perform arithmetic multiplication

Format:

```
opcode mulr.rx, prod.mx                  2 operand
opcode mulr.rx, muld.rx, prod.wx         3 operand
```

Operation:

```
prod ←prod * mulr;                       !2 operand
prod ←muld * mulr;                       !3 operand
```

Condition Codes:

```
N ← prod LSS 0;
Z ← prod EQL 0;
V ← overflow;
C ← 0;
```

Exceptions:

Integer overflow
Floating overflow
Floating underflow
Reserved operand

Opcodes:

84	MULB2	Multiply Byte 2 Operand
85	MULB3	Multiply Byte 3 Operand
A4	MULW2	Multiply Word 2 Operand
A5	MULW3	Multiply Word 3 Operand
C4	MULL2	Multiply Longword 2 Operand
C5	MULL3	Multiply Longword 3 Operand
44	MULF2	Multiply F_floating 2 Operand
45	MULF3	Multiply F_floating 3 Operand
64	MULD2	Multiply D_floating 2 Operand
65	MULD3	Multiply D_floating 3 Operand
44FD	MULG2	Multiply G_floating 2 Operand
45FD	MULG3	Multiply G_floating 3 Operand
64FD	MULH2	Multiply H_floating 2 Operand
65FD	MULH3	Multiply H_floating 3 Operand

Description: In 2-operand format, the product operand is multiplied by the multiplier operand and the product operand is replaced by the result. In 3-operand format, the multiplicand operand is multiplied by the multiplier operand and the product operand is replaced by the result. In floating format, the product operand result is rounded for both 2- and 3-operand format.

Notes:

1. Integer overflow occurs if the high half of the double-length result is not equal to the sign extension of the low half.
2. On a floating reserved operand fault, the product operand is unaffected and the condition codes are unpredictable.

3. On floating underflow and floating overflow, the subsequent action varies dependent upon which VAX family processor is being used.

Example: MULTIPLY FLOATING 2 OPERAND

```
MULF2 R8, R7            ;Multiply floating contents
                        ;of R8 by contents
                        ;of R7; store
                        ;result in R7
```

Initial Conditions:
R8 = 00004220
R7 = 00004410

After Instruction Execution:
R8 = 00004220
R7 = 000045B4

Example: MULTIPLY FLOATING 3 OPERAND

```
MULF3 R8, R7, R0     ;Multiply floating contents
                     ;of R8 by contents
                     ;of R7; store result
                     ;in R0
```

Initial Conditions:
R8 = 00004220
R7 = 000045B4
R0 = 00004410

After Instruction Execution:
R8 = 00004220
R7 = 000045B4
R0 = 00004761

EMUL

EXTENDED MULTIPLY

Purpose: perform extended-precision multiplication

Format: opcode mulr.rl, muld.rl, add.rl, prod.wq

Operation: prod ← {muld * mulr} + SEXT(add)

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

Exceptions: None

Opcodes: 7A EMUL Extended Multiply

Description: The multiplicand operand is multiplied by the multiplier operand giving a double length result. The addend operand is sign-extended to double length and added to the result, and then the product operand is replaced by the final result.

Notes:

Example: EXTENDED MULTIPLY

To multiply two quadwords, producing a quadword;

```
      EMUL A, B, #0, C      ;multiply low half
      MULL3 A+4, B, R0      ;high half = A [high] *
                            ;B [low]
      MULL3 A, B+4, R1      ;+ A [low] * B
                            ;[high]
      ADDL R1, R0           ;(combine)
      TSTL A                ;if A [low] < 0, need to
      BGEQ 10$              ;compensate for
                            ;unsigned
      ADDL B, R0            ;bias of 2**32
10$:TSTL B                  ;if B [low] <0, need to
      BGEQ 20$              ;compensate for
                            ;unsigned
                            ;bias of 2**32
      ADDL A, R0
20$:ADDL R0, C+4            ;combine with high half
                            ;of A [low] * B [low]
```

EMOD

EXTENDED MULTIPLY AND INTEGERIZE

Purpose: perform accurate range reduction of math function arguments

Format:
EMODF and EMODD
opcode mulr.rx, mulrx.rb, muld.rx, int.wl, fract.wx
EMODG and EMODH
opcode mulr.rx, mulrx.rw, muld.rx, int.wl, fract.wx.

Operation:
int ← integer part of muld * {mulr'mulrx};
fract ← fractional part of muld *
{mulr'mulrx};

Condition Codes:
N ← fract LSS 0;
Z ← fract EQL 0;
V ← {integer overflow};
C ← 0

Exceptions:
Integer overflow
Floating underflow
Reserved operand

Opcodes:

54	EMODF	Extended Multiply and Integerize F_floating
74	EMODD	Extended Multiply and Integerize D_floating
54FD	EMODG	Extended Multiply and Integerize G_floating
74FD	EMODH	Extended Multiply and Intergerize H_floating

Description: The multiplier extension operand is concatenated with the multiplier operand to gain 8 (EMODD and EMODF), 11 (EMODG), or 15 (EMODH) additional low-order fraction bits. The low-order 5 or 1 bits of the 16-bit multiplier extension operand are ignored by the EMODG and EMODH instructions, respectively. The multiplicand operand is multiplied by the extended multiplier operand. This multiplication is such that the result is equivalent to the exact product truncated (before normalization) to a fraction field of 32 bits in F_floating, 64 bits in D_floating and G_floating, and 128 in H_floating. Regarding the result as the sum of an integer and fraction of the same sign, the integer operand is replaced by the integer part of the result and the fraction operand is replaced by the rounded fractional part of the result.

Notes:

1. On a reserved operand fault, the integer operand and the fraction operand are unaffected. The condition codes are unpredictable.
2. On floating underflow, the sequence of events depends upon the VAX processor being used.

3. On integer overflow, the integer operand is replaced by the low order bits of the true result.
4. Floating overflow is indicated by integer overflow; however, integer overflow is possible in the absence of floating overflow.
5. The signs of the integer and fraction are the same unless integer overflow results.
6. Because the fraction part is rounded after separation of the integer part, it is possible that the value of the fraction operand is 1.

DIV

DIVIDE

Purpose: perform arithmetic division

Format:

opcode divr.rx, quo.mx		2 operand
opcode divr.rx, divd.rx, quo.wx		3 operand

Operation:

quo ← quo / divr;		!2 operand
quo ← divd / divr;		!3 operand

Condition Codes:

N ← quo LSS 0;
Z ← quo EQL 0;
V ← {overflow} OR {divr EQL 0};
C ← 0

Exceptions:

Integer overflow
Divide by zero
Floating overflow
Floating underflow
Reserved operand

Opcodes:

86	DIVB2	Divide Byte 2 Operand
87	DIVB3	Divide Byte 3 Operand
A6	DIVW2	Divide Word 2 Operand
A7	DIVW3	Divide Word 3 Operand
C6	DIVL2	Divide Longword 2 Operand
C7	DIVL3	Divide Longword 3 Operand
46	DIVF2	Divide F_floating 2 Operand
47	DIVF3	Divide F_floating 3 Operand
66	DIVD2	Divide D_floating 2 Operand
67	DIVD3	Divide D_floating 3 Operand
46FD	DIVG2	Divide G_floating 2 Operand
47FD	DIVG3	Divide G_floating 3 Operand
66FD	DIVH2	Divide H_floating 2 Operand
67FD	DIVH3	Divide H_floating 3 Operand

Description: In 2-operand format, the quotient operand is divided by the divisor operand and the quotient operand is replaced by the result. In 3-operand format, the dividend operand is divided by the divisor operand and the quotient operand is replaced by the result. In floating format, the quotient operand result is rounded for both 2- and 3-operand format.

Notes:

1. Integer division is performed so that the remainder (unless it is zero) has the same sign as the dividend; i.e., the result is truncated towards 0.
2. Integer overflow occurs if and only if the largest negative integer is divided by −1. On overflow, operands are affected as in 3 below.

3. If the integer divisor operand is 0, then in 2-operand integer format, the quotient operand is not affected; in 3-operand format the quotient operand is replaced by the dividend operand.
4. On a floating reserved operand fault, the quotient operand is unaffected and the condition codes are unpredictable.
5. On floating underflow and floating overflow, the subsequent actions depend upon the VAX family processor being used.
6. Floating divide by zero, similarly, produces various actions according to the processor being used.

Example:

```
DIVIDE FLOATING 2 OPERAND
DIVF2 R4, R2                          ;Divide
```

Initial Conditions:

R4 = 00004100
R2 = 00004330

After Instruction Execution:

R4 = 00004100
R2 = 000042B0

EDIV

EXTENDED DIVIDE

Purpose: perform extended-precision division

Format: opcode divr.rl, divd.rq, quo.wl, rem.wl

Operation:

```
quo ← divd/divr;
rem ← REM(divd, divr)
```

Condition Codes:

```
N ← quo LSS 0;
Z ← quo EQL 0;
V ← {integer overflow} OR {divr EQL 0};
C ← 0;
```

Exceptions: Integer overflow
Divide by zero

Opcodes: 7B EDIV Extended Divide

Description: The dividend operand is divided by the divisor operand; the quotient operand is replaced by the quotient and the remainder operand is replaced by the remainder.

Notes:

1. The division is performed so that the remainder operand (unless it is 0) has the same sign as the dividend operand.
2. On overflow, the operands are affected as in Note 3, below.
3. If the divisor operand is 0, then the quotient operand is replaced by bits 31:0 of the dividend operand, and the remainder is replaced by 0.

Example:

```
MOVQ       #53, B
MOV L      #10, A
CLRL       C
CLRL       D
EDIV       A, B, C, D
```

The result of the extended division is that A = 10 (unchanged), B = 53 (unchanged), C = 5 (quotient of division), and D = 3 (remainder).

BIT TEST

Purpose: test a set of bits for all zero

Format: opcode mask.rx, src.rx

Operation: tmp ← src AND mask;

Condition Codes:

```
N ← tmp LSS 0;
Z ← tmp EQL 0;
V ← 0;
C ← C
```

Exceptions: None

Opcodes:

93	BITB	Bit Test Byte
B3	BITW	Bit Test Word
D3	BITL	Bit Test Long

Description: The mask operand is ANDed with the source operand. Both operands are unaffected. The only action is to affect condition codes.

BIS

BIT SET

Purpose: perform logical inclusive OR of two integers

Format:

```
opcode mask.rx, dst.mx              2 operand
opcode mask.rx, src.rx, dst.wx      3 operand
```

Operation:

```
dst ← dst OR mask;                  !2 operand
dst ← src OR mask;                  !3 operand
```

Condition Codes:

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

Exceptions: None

Opcodes:

88	BISB2	Bit Set Byte 2 Operand
89	BISB3	Bit Set Byte 3 Operand
A8	BISW2	Bit Set Word 2 Operand
A9	BISW3	Bit Set Word 3 Operand
C8	BISL2	Bit Set Longword 2 Operand
C9	BISL3	Bit Set Longword 3 Operand

Description: In 2-operand format, the mask operand is ORed with the destination operand and the destination operand is replaced by the result. In 3-operand format, the mask operand is ORed with the source operand and the destination operand is replaced by the result.

BIC

BIT CLEAR

Purpose: perform complemented AND of two integers

Format:

opcode mask.rx, dst.mx   2 operand
opcode mask.rx, src.rx, dst.wx   3 operand

Operation:

dst ← dst AND {NOT mask};   !2 operand
dst ← src AND {NOT mask};   !3 operand

Condition Codes:

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

Exceptions: None

Opcodes:

8A	BICB2	Bit Clear Byte 2 operand
8B	BICB3	Bit Clear Byte 3 operand
AA	BICW2	Bit Clear Word 2 operand
AB	BICW3	Bit Clear Word 3 operand
CA	BICL2	Bit Clear Longword 2 operand
CB	BICL3	Bit Clear Longword 3 operand

Description: In 2-operand format, the destination operand is ANDed with the one's complement of the mask operand and the destination operand is replaced by the result. In 3-operand format, the source operand is ANDed with the one's complement of the mask operand and the destination operand is replaced by the result.

XOR

EXCLUSIVE OR

Purpose: perform logical exclusive OR of two integers

Format:

```
opcode mask.rx, dst.mx             2 operand
opcode mask.rx, src.rx, dst.wx     3 operand
```

Operation:

```
dst ← dst XOR mask;                !2 operand
dst ← src XOR mask;                !3 operand
```

Condition Codes:

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

Exceptions: None

Opcodes:

8C	XORB2	Exclusive OR Byte 2 Operand
8D	XORB3	Exclusive OR Byte 3 Operand
AC	XORW2	Exclusive OR Word 2 Operand
AD	XORW3	Exclusive OR Word 3 Operand
CC	XORL2	Exclusive OR Longword 2 Operand
CD	XORL3	Exclusive Or Longword 3 Operand

Description: In 2-operand format, the mask operand is XORed with the destination operand and the destination operand is replaced by the result. In 3-operand format, the mask operand is XORed with the source operand and the destination operand is replaced by the result.

ASH

ARITHMETIC SHIFT

Purpose: shift of integer

Format: opcode cnt.rb, src.rx, dst.wx

Operation: dst ← src shifted cnt bits;

Condition Codes:
N ← dst LSS 0;
Z ← dst EQL 0;
V ← {integer overflow};
C ← 0

Exceptions: Integer overflow

Opcodes:

78	ASHL	Arithmetic Shift Longword
79	ASHQ	Arithmetic Shift Quadword

Description: The source operand is arithmetically shifted by the number of bits specified by the count operand, and the destination operand is replaced by the result. The source operand is unaffected. A positive count operand shifts to the left bringing 0s into the least significant bit; a negative count operand shifts to the right bringing in copies of the most significant (sign) bit into the most significant bit position. A zero count operand replaces the destination operand with the unshifted source operand.

Notes:

1. Integer overflow occurs on a left shift if any bit shifted into the sign bit position differs from the sign bit of the source operand.
2. If cnt GTR 32 (ASHL) or cnt GTR 64 (ASHQ), the destination operand is replaced by 0.
3. If cnt LEQ −32 (ASHL) or cnt LEQ −63 (ASHQ), all the bits of the destination operand are copies of the sign bit of the source operand.

ROTL

ROTATE LONG

Purpose: rotate integer

Format: opcode cnt.rb, src.rl,dst.wl

Operation: dst ← src rotated cnt bits;

Condition Codes:

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

Exceptions: None

Opcodes:

9C	ROTL	Rotate Longword

Description: The source operand is rotated logically by the number of bits specified by the court operand and the destination operand is replaced by the result. The source operand is unaffected. A positive count operand rotates to the left. A negative count operand rotates to the right. A 0 count operand replaces the destination operand with the source operand.

POLY

POLYNOMINAL EVALUATION

Purpose: allows fast calculation of math functions

Format: opcode arg.rx, degree.rw, tbladdr.ab

Operation:

```
tmp1 ← degree;
if tmp1 GRTU 31 then RESERVED OPERAND EXCEPTION;
tmp2 ← tbladdr;
tmp3 ← {(tmp2);           !tmp3 accumulates the
                          partial result
                          !tmp3 is of type X
if POLYH then −(SP) ← arg;
tmp4 ← 0;                 !underflow flag for
                          original 11/780
while tmp1 GRTU 0 do
   begin                  !computation loop
   tmp5 ← {arg * tmp3};   !tmp5 accumulates new partial
                          result.
                          !tmp3 has old partial result.
                          !Perform multiply, and
                          retain the 31 (POLYF),
                          !63 (POLYD, POLYG), or 127
                          (POLYH) most significant
                          !bits of the fraction by
                          truncating the unnormalized
                          !product. (The most significant
                          bit of the 31, 63,
                          !or 127 bits in the
                          product magnitude will be zero
                          !if the product magnitude is
                          LSS ½ and GEQ 1/4.)
                          !Use the result in the
                          following add operation.

   tmp5 ← tmp5 + (tmp2);
                          !normalize, and round to
                             type X.
                          !Check for over/underflow
                             only after the combined
                          !multiply/add/normalize/
                             round sequence.
if OVERFLOW then FLOATING OVERFLOW EXCEPTION
if UNDERFLOW then
      begin
      if FU EQL 1 then FLOATING UNDERFLOW FAULT;
                          !except for original
                          VAX-11/780
tmp5 ← 0;                 !force result to 0;
```

```
          if FU EQL 1 then tmp4 ← 1;
                                  !set underflow flag
                                  (original 11/780)
          end;
     tmp1 ← tmp1 - 1;
     tmp2 ← tmp2 + {size of data type};
     tmp3 ← tmp5;
                                  !update partial result in tmp3
     end;
if POLYF then
     begin
     R0 ← tmp3;
     R1 ← 0;
     R2 ← 0;
     R3 ← tmp2;
     end;
if POLYD or POLYG then
     mbegin
     R1'R0 ← tmp3;
     R2 ← 0;
     R3 ← tmp2;
     R4 ← 0;
     R5 ← 0;
     end;
if POLYH then
     begin
     SP ← SP + 16;
     R3'R2'R1'R0 ← tmp3;
     R4 ← 0;
     R5 ← tmp2;
     end;
if tmp4 EQL 1 then FLOATING UNDERFLOW TRAP !original
11/780
                                 only
```

Condition Codes:

N ← R0 LSS 0;
Z ← R0 EQL 0;
V ← {floating overflow};
C ← 0;

Exceptions:

Floating overflow
Floating underflow
Reserved operand

Opcodes:

55	POLYF	Polynomial Evaluation F_floating
75	POLYD	Polynomial Evaluation D_floating
55FD	POLYG	Polynomial Evaluation G_floating
75FD	POLYH	Polynomial Evaluation H_floating

Description: The table address operand points to a table of polynomial coefficients; the coefficient of the highest order term of the polynomial is pointed to by the table address operand. The table is specified with lower order coefficients stored at increasing addresses. The data type of the coefficients is the same as the data type of the argument operand.

Evaluation is carried out by Horner's method, and the contents of R0 (R1'R0 for POLYD and POLYG, R3'R2'R1'R0 for POLYH) are replaced by the result. The result computed is:
result = C[0] + x*(C[1] + x*(C[2] +...x*C [d])), where d = degree and x = arg.

The unsigned word degree operand specifies the highest numbered coefficient to participate in the evaluation. POLYH requires four longwords on the stack to store arg in case the instruction is interrupted.

Notes:

1. After execution:

   POLYF
   R0 = result
   R1 = 0
   R2 = 0
   R3 = table address + degree*4 + 4

   POLYD
   R0 = high order part of result
   R1 = low order part of result
   R2 = 0
   R3 = table address + degree*8 + 8
   R4 = 0
   R5 = 0

   POLYH
   R0 = highest order part of the result
   R1 = second highest order part of the result
   R2 = second lowest order part of the result
   R3 = lowest order part of the result
   R4 = 0
   R5 = table address + degree*16 + 16

2. On a floating fault

   - If PSL<FPD> = 0, the instruction faults and all relevant side effects are restored to their original state.
   - If PSL<FPD> = 1, the instruction is suspended and state is saved in the general registers as follows:

     POLYF
     R0 = tmp3 !partial result after iteration prior to
     !the one causing the overflow/underflow

```
R1 = arg
R2<7:0> = tmp1          !number of iterations
          remaining
R2<31:8> = implementation specific
R3 = tmp2     !points to table entry causing excep-
tion

POLYD and POLYG
R1'R0 = tmp3
          !partial result after iteration prior to the
          !one causing the overflow/underflow
R2<7:0> = tmp1 !number of iterations remaining
R2<31:8> = implementation specific
R3 = tmp2     !points to table entry causing excep-
tion
R5'R4 = arg

POLYH
R3'R2'R1'R0 = tmp3      !partial result after iteration
prior to
          !the one causing the overflow/underflow
R4<7:0> = tmp1     !number of iterations remain-
ing
R4<31:8> = implementation specific
R5 = tmp2           !points to table entry causing
exception
arg is on previous mode stack.
```

Implementation specific information is saved to allow the instruction to continue after possible scaling of the coefficients and partial result by a fault handler.

3. If the unsigned word degree operand is 0, the result is c[0].
4. If the unsigned word degree operand is greater than 31, a reserved operand exception occurs.
5. On a reserved operand exception:
   - If PSL<FPD> = 0, the reserved operand is either the degree operand (greater than 31), or the argument operand, or some coefficient.
   - If PSL<FPD> = 1, the reserved operand is a coefficient, and R3 (except for POLYH) or R5 (for POLYH) is pointing at the value which caused the exception.

- The state of the saved condition codes and the other registers is unpredictable. If the reserved operand is changed and the contents of the condition codes and all registers are preserved, the fault is continuable.

6. On floating underflow after the rounding operation, the subsequent action is dependent upon the VAX family processor being used.
7. On floating overflow after the rounding operation at any iteration of the computation loop, the instruction terminates and causes a trap or fault (processor dependent). On overflow traps, the contents of R2 and R3 are unpredictable for POLYF, POLYD, POLYG, and 0 for POLYH; the contents of R4 and R5 are unpredictable for POLYD, POLYG, and POLYH; R0 contains the reserved operand (minus 0) and R1 = 0.
8. POLY can have both overflow and underflow in the same instruction. If both occur, overflow trap is taken; underflow is lost.
9. If the argument is zero and one of the coefficients in the table is the reserved operand, whether a reserved operand fault occurs is unpredictable.
10. For POLYH, some implementation may not push arg on the stack until after an interrupt or fault occurs. However, arg will always be on the stack if an interrupt of floating fault occurs after FPD is set.

Example

To compute P(x) = C0 + C1*x+C2*x**2
where C0 = 1.0, C1 = .5, and C2 = .25

```
                POLYF   X,#2,PTABLE
                .
                .
                .
PTABLE:
                .FLOAT  0.25    ;C2
                .FLOAT  0.5     :C1
                .FLOAT  1.0     ;C0
```

CHAPTER 12
SPECIAL INSTRUCTIONS

INTRODUCTION

Most of the instructions in this chapter are optimizations of frequently occurring sequences of code. Your programmers can increase their accuracy, efficiency, and productivity by exploiting these prebuilt aids that DIGITAL supplies. Included among them are instructions that manipulate the multiple registers, the Processor Status Longword, addresses, indices, queues, and variable length bit fields.

MULTIPLE REGISTER INSTRUCTIONS

Multiple register instructions allow the saving and restoring of multiple registers in one operation. In either case, the save area is on the stack; the PUSHR instruction saves multiple registers by pushing them onto the stack; the POPR instruction restores multiple registers by popping them from the stack. A 16-bit mask operand, with bit n representing Rn, specifies the list of registers. This mask is a normal read operand, so it can be calculated or it can be an in-line literal. When only registers in the range R0 through R5 are being saved or restored, the mask can be expressed as a short literal.

The software standard for calling and signaling requires that registers be saved in the call frame. Thus, any registers manipulated by PUSHR or POPR, except R0 and R1, must appear in the procedure entry mask. The standard also requires any registers between R2 and R11 which are modified by the procedure to be saved in the call frame by setting up the appropriate entry mask. R0 and R1 are used to return procedure status. PUSHR/POPR should be used to save and restore only those registers specified in the procedure entry mask, for if a procedure saves registers not noted in the entry mask and it gets an exception, its caller's registers cannot be restored properly by the unwinding mechanism.

PUSHR

PUSH REGISTERS

Purpose: save multiple registers on stack

Format: opcode mask.rw

Operation:

```
for tmp ← 14 step −1 until 0 do
if mask <tmp> EQL 1 then −(sp) ← R[tmp];
```

Condition Codes:

N ← N;
Z ← Z;
V ← V;
C ← C

Exceptions: None

Opcodes:

BB	PUSHR	Push Registers

Description: The contents of registers whose number corresponds to set bits in the mask operand are pushed on the stack as longwords. R[n] is pushed if mask <n> is set. The mask is scanned from bit 14 to bit 0, and bit 15 is ignored.

Notes: The order of pushing is specified so that the contents of higher numbered registers are stored at higher memory addresses. This results in, say, a D_floating datum stored in adjacent registers being stored by PUSHR in memory in the correct order.

This instruction is similar to the sequence

```
PUSHL    R14
PUSHL    R13
  .
  .
  .
PUSHL    R0
```

where only the masked registers are pushed.

Example:

```
PUSHR #↑M<R0,R1,R2,R3>          ;saves R0
                                ;through R3
```

Example:

```
PUSHR #↑M<R1,R6,R7>        ;saves R1, R6, and R7
```

POPR

POP REGISTERS

Purpose: restore multiple registers from stack

Format: opcode mask.rw

Operation:

```
for tmp ← 0 step 1 until 14 do
if mask <tmp> EQL 1 then R[tmp] ← (SP)+;
```

Condition Codes:

```
N ← N;
Z ← Z;
V ← V;
C ← C
```

Exceptions: None

Opcodes: BA   POPR   Pop Registers

Description: The contents of registers whose number corresponds to set bits in the mask operand are replaced by longwords popped from the stack. R[n] is replaced if mask <n> is set. The mask is scanned from bit 0 to bit 14, with bit 15 ignored.

Notes: This instruction is similar to the sequence

```
        MOVL    (SP)+,R0
        MOVL    (SP)+,R1
          .
          .
          .
        MOVL    (SP)+,R14
```

where only the masked registers are popped.

PROCESSOR STATUS LONGWORD MANIPULATION

MOVPSL

MOVE FROM PSL

Purpose: obtain processor status

Format: opcode dst.wl

Operation: dst ← PSL;

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

Exceptions: None

Opcodes:

DC	MOVPSL	Move from PSL

Description: The destination operand is replaced by the Processor Status Longword.

BISPSW
BICPSW

BIT SET PSW
BIT CLEAR PSW

Purpose: set or clear trap enables

Format: opcode mask.rw

Operation:

```
PSW ← PSW OR mask;                          !BISPSW
PSW ← PSW AND {NOT mask};                   !BICPSW
```

Condition Codes:

```
N ← N OR mask <3>;                          !BISPSW
Z ← Z OR mask <2>;
V ← V OR mask <1>;
C ← C OR mask <0>

N ← N AND {NOT mask} <3>;                   !BICPSW
Z ← Z AND {NOT mask} <2>;
V ← V AND {NOT mask} <1>;
C ← C AND {NOT mask} <0>
```

Exceptions: Reserved Operand

Opcodes:

B8	BISPSW	Bit set PSW
B9	BICPSW	Bit clear PSW

Description: On BISPSW, the Processor Status Longword is ORed with the 16-bit mask operand and the PSW is replaced by the result. On BICPSW, the Processor Status Longword is ANDed with the 1's complement of the 16-bit mask operand and the PSW is replaced by the result.

Notes: A reserved operand fault occurs if mask <15:8> is not zero. On a reserved operand fault, the PSW is not affected.

Example:

```
BISPSW     #↑M<FU>              ;enables floating
                                ;underflow traps
```

ADDRESS INSTRUCTIONS

MOVA
PUSHA

MOVE ADDRESS
PUSH ADDRESS

Purpose: calculate address of quantity

Format:

```
opcode src.ax, dst.wl          !MOVA
opcode src.ax                  !PUSHA
```

Operation:

```
dst ← src;                     !MOVA
−(SP) ← src;                   !PUSHA
```

Condition Codes:

```
N ← dst LSS 0;                 !MOVA
Z ← dst EQL 0;
V ← 0;
C ← C

N ← src LSS 0;                 !PUSHA
Z ← src EQL 0;
V ← 0;
C ← C
```

Exceptions: None

Opcodes:

9E	MOVAB	Move Address Byte
3E	MOVAW	Move Address Word
DE	MOVAL	Move Address Longword
	MOVAF	Move Address F_Floating
7E	MOVAQ	Move Address Quadword
	MOVAD	Move Address D_Floating
	MOVAG	Move Address G_Floating
7EFD	MOVAH	Move Address H_Floating
	MOVAO	Move Address Octaword
9F	PUSHAB	Push Address Byte
3F	PUSHAW	Push Address Word
DF	PUSHAL	Push Address Longword
	PUSHAF	Push Address F_Floating
7F	PUSHAQ	Push Address Quadword
	PUSHAD	Push Address D_Floating
	PUSHAG	Push Address G_Floating
7FFD	PUSHAH	Push Address H_Floating
	PUSHAO	Push Address Octaword

Description: For MOVA, the destination operand is replaced by the source operand, which is an address. For PUSHA, the source operand is pushed on the stack. The context in which the source operand is evaluated is given by the data type of the instruction. The operand whose address replaces the destination operand is not referenced.

Notes:

1. The source operand is of address access type which causes the address of the specified operand to be moved.
2. PUSHAx is equivalent to MOVAx src, −(SP), but is shorter.
3. The only difference between the MOVAxs (PUSHxs) is the context of the src. This only affects autoincrement, autodecrement, and indexing.

Example:

```
PUSHAL XYZ      ;pushes the address of
                ;longword XYZ
```

INDEX INSTRUCTION

The index instruction (INDEX) calculates an index for an array of fixed length data types (integer and floating) and for arrays of bit fields, character strings, and decimal strings. It accepts as arguments: a subscript, lower and upper subscript bounds, an array element size, a given index, and a destination for the calculated index. It incorporates range checking within the calculation for high-level languages using subscript bounds, and it aids index calculation optimization by removing invariant expressions.

INDEX

COMPUTE INDEX

Purpose: index calculation of arrays of fixed length data, bit fields, and strings

Format: opcode subscript.rl, low.rl, high.rl, size.rl, indexin.rl, indexout.wl

Operation:

```
indexout ← {indexin + subscript}*size;
if {subscript LSS low} or {subscript GTR high}
then {subscript range trap};
```

Condition Codes:

```
N ← indexout LSS 0;
Z ← indexout EQL 0;
V ← 0;
C ← 0
```

Exceptions: Subscript range

Opcodes:

0A	INDEX	Index

Description: The indexin operand is added to the subscript operand and the sum is multiplied by the size operand. The indexout operand is replaced by the result. If the subscript operand is less than the low operand or greater than the high operand, a subscript range trap is taken.

Notes:

1. No arithmetic exception other than subscript range can result from this instruction. Thus, no indication is given if overflow occurs in either the add or multiply steps. If overflow occurs on the add step the sum is the low order 32 bits of the true result. If overflow occurs on the multiply step the indexout operand is replaced by the low order 32 bits of the true product of the sum and the subscript operand. In the normal use of this instruction, overflow cannot occur without a subscript range trap occurring.

2. The Index instruction is useful in index calculations for arrays of the fixed length data types (integer and floating) and for index calculations for arrays of bit fields, character strings, and decimal strings. The indexin operand permits cascading Index instructions for multidimensional arrays. For one-dimensional bit field arrays it also permits introduction of the constant portion of an index calculation which is not readily absorbed by address arithmetic.

Example: The COBOL statements:

```
01          A-ARRAY.
                  02      A PIC x(10) occurs 15 times
01          B PIC x(10).
            MOVE A(I) to B.
```

are equivalent to:

```
            INDEX I, #1, #15, #10, #0, R0
            MOVC3 #10, A-10[R0], B.
```

The PL/I statements:

```
DCL A(−3:10) Bit (5);
A(I) = 1;
```

are equivalent to:

```
            INDEX I, #−3, #10, #5, #3, R0
            INSV #1, R0, #5, A; assumes A
            byte-aligned
```

The FORTRAN statements:

```
INTEGER*4 A(L1:U1, L2:U2), I, J
A(I,J) = 1
```

are equivalent to:

```
            INDEX J, #L2, #U2, #M1, #0, R0;
            M1=U1−L1+1
            INDEX I, #L1, #U1, #1, R0, R0;
            MOVL #1, A-a[R0]; a={{L2*M1}+L1}*4
```

QUEUE INSTRUCTIONS

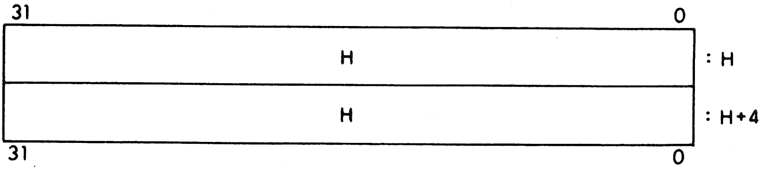
A queue is a circular, doubly linked list whose entries are specified by their addresses. Each queue entry links to two others via a pair of longwords. The first (lower addressed) is the forward link (FLINK); it specifies the location of the succeeding entry; the second, the backward link (BLINK), specifies the location of the preceding entry. Two distinct types of queues are possible in VAXes—**absolute** and **self-relative**—classified according to the type of links they use. An absolute link contains the absolute address of the entry that it points to, while a self-relative link contains a displacement from the present queue entry.

Absolute Queues

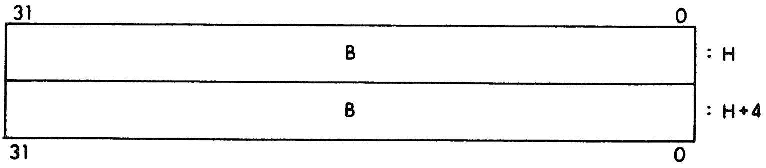
An absolute queue is specified by a queue header which is identical to a pair of queue linkage longwords. The forward link of the header is the address of the entry termed the head of the queue. The backward link of the header is the address of the entry termed the tail of the queue. The forward link of the tail points to the header.

Two general operations can be performed on queues: insertion of entries and removal of entries. Generally, entries can be inserted or removed only at the head or tail of a queue.

The following examples illustrate some queue operations. An empty queue is specified by its header at address H:



If an entry at address B is inserted into an empty queue (at either the head or tail) the queue is as shown below:



31 0	
H	: B
H	: B+4
31 0	

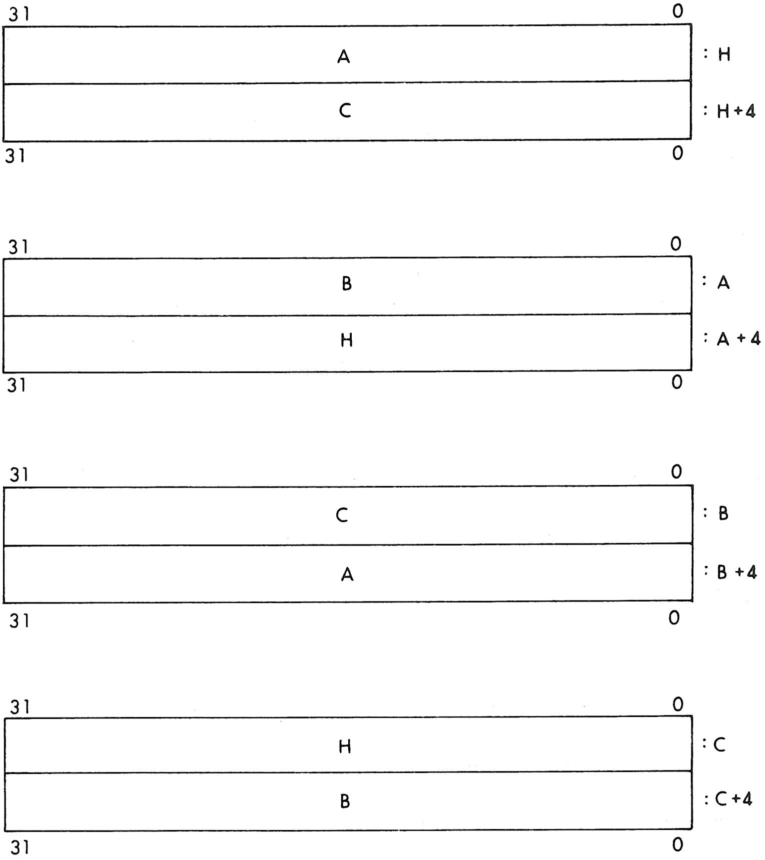
If an entry at address A is inserted at the head of the queue, the queue is as shown below:

31 0	
A	: H
B	: H +4
31 0	

31 0	
B	: A
H	: A +4
31 0	

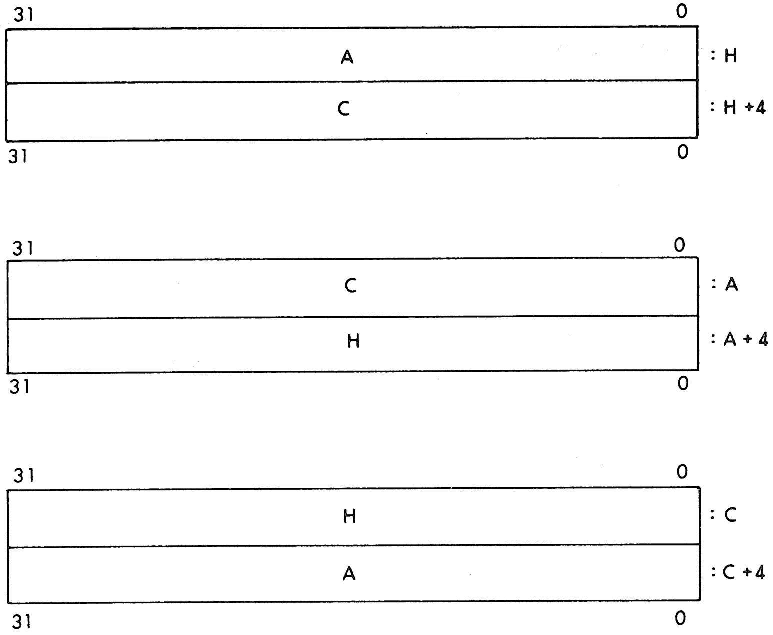
31 0	
H	: B
A	: B +4
31 0	

Finally, if an entry at address C is inserted at the tail, the queue appears as follows:



Following the above steps in reverse order gives the effect of removal at the tail and removal at the head.

If more than one process can perform operations on a queue simultaneously, insertions and removals should only be done at the head or tail of the queue, but when just one process (or one process at a time) can perform operations on a queue, insertions and removals can be made at other locations. In the above example with the queue containing entries A, B, and C, the entry at address B can be removed, giving:



The reason for the above restriction is that operations at the head or tail are always valid because the queue header is always present; operations elsewhere in the queue depend on specific entries being present and may become invalid if another process is concurrently performing operations on the queue.

Two instructions are provided for manipulating absolute queues: INSQUE, and REMQUE. INSQUE inserts an entry specified by an entry operand into the queue, following the entry specified by the predecessor operand. REMQUE removes the entry specified by the entry operand. Queue entries can be on arbitrary byte boundaries. Both INSQUE and REMQUE are implemented as noninterruptible instructions.

INSQUE

INSERT ENTRY IN QUEUE

Purpose: add entry to head or tail of queue

Format: opcode entry.ab, pred.ab

Operation:

```
If {all memory accesses can be completed} then
   begin
(entry) <- (pred);                       !forward link of entry
(entry + 4) <- pred;                     !backward link of entry
((pred) + 4) <- entry;                   !backward link of
                                         !successor
(pred) <- entry;                         !forward link of
                                         !predecessor
end;
else
begin
{backup instruction};
{initiate fault};
end;
```

Condition Codes:

```
N <- (entry) LSS (entry + 4);
Z <- (entry) EQL (entry + 4);           !first entry
                                        !in queue
V <- 0;
C <- (entry) LSSU (entry + 4)
```

Exceptions: None

Opcodes:

0E	INSQUE	Insert Entry in Queue

Description: The entry specified by the entry operand is inserted into the queue following the entry specified by the predecessor operand. If the entry inserted was the first one in the queue, the condition code Z-bit is set; otherwise it is cleared. The insertion is a noninterruptible operation. Before performing any part of the operation, the processor validates that the entire operation can be completed. This ensures that if a memory management exception occurs, the queue is left in a consistent state.

Notes:

1. Because the insertion is noninterruptible, processes running in kernel mode can share queues with interrupt service routines.
2. The INSQUE instruction is implemented such that cooperating software processes in a single processor may access a shared list without additional synchronization if the insertions and removals are only at the head or trail of the queue.

3. During access validation, any access which cannot be completed results in a memory management exception, even though the queue insertion is not started..

Example: Three types of insertion can be performed by appropriate choice of predecessor operand:

Insert at head

```
INSQUE     entry,h                    ;h is queue head
```

Insert at tail

```
INSQUE     entry,@h+4                 ;h is queue head
```

(Note "@" in this case only)

Insert after arbitrary predecessor

```
INSQUE     entry,p                    ;p is predecessor
```

To set a software interlock realized with a queue, the following can be used:

```
      INSQUE ...                      ;was queue empty?
      BEQL      1$                    ;yes
      CALL      WAIT(...)             ;no, wait
1$:
```

REMQUE

REMOVE ENTRY FROM QUEUE

Purpose: remove entry from head or tail of queue

Format: opcode entry.ab, addr.wl

Operation:

```
if {all memory accesses can be completed} then
   begin
   ((entry+4)) ← (entry);          !forward link of predecessor
   ((entry)+4) ← (entry+4);        !backward link of successor
   addr ← entry;
   end;
else
   begin
   {backup instruction};
   {initiate fault};
   end;
```

Condition Codes:

```
N ← (entry) LSS (entry+4);
Z ← (entry) EQL (entry+4);                 !queue empty
V ← entry EQL (entry+4);             !no entry to remove
C ← (entry) LSSU (entry+4)
```

Exceptions: None

Opcodes: 0F REMQUE Remove Entry from Queue

Description: The queue entry specified by the entry operand is removed from the queue. The address operand is replaced by the address of the entry removed. If there was no entry in the queue to be removed, the condition code V bit is set; otherwise it is cleared. If the queue is empty at the end of this instruction, the condition code Z-bit is set; otherwise it is cleared. The removal is a noninterruptible operation. Before performing any part of the operation, the processor validates that the entire operation can be completed. This ensures that if a memory management exception occurs, the queue is left in a consistent state.

Notes:

1. Because the removal is noninterruptible, processes running in kernel mode can share queues with interrupt service routines.
2. The REMQUE instruction is implemented such that cooperating software processes in a single processor may access a shared list without additional synchronization if insertions and removals are only at the head or tail of the queue.
3. During access validation, any access which cannot be completed results in a memory management exception, even though the queue removal is not started.

Example: Three types of removal can be performed by suitable choice of entry operand:

```
Remove at head
REMQUE       @h,addr              ;h is queue header

Remove at tail
REMQUE       @h+4, addr           ;h is queue header

Remove arbitrary entry
REMQUE       entry,addr           ;
```

To release a software interlock realized with a queue, the following can be used:

```
    REMQUE ...                    ;queue empty?
    BEQL    1$                    ;yes
    CALL    ACTIVATE (...)        ;activate other
                                  waiters
1$:
```

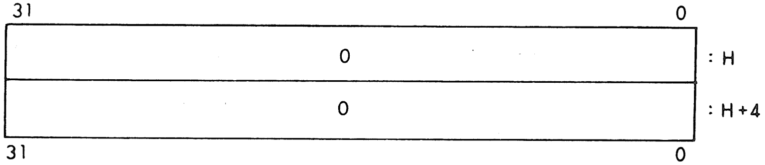
To remove entries until the queue is empty, the following can be used:

```
1$: REMQUE...          ;anything removed?
    BVS   EMPTY        ;no
    .
    .
    .
    BR   1$            ;
```

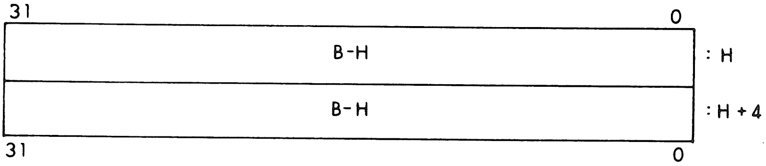
Self-Relative Queues

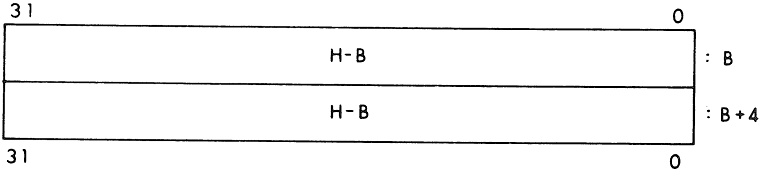
Self-relative queues use displacements from queue entries as links. As with absolute queues, queue entries are linked by a pair of longwords. The first (lower addressed) is the forward link—displacement of the succeeding queue entry from the present entry. The second longword (higher addressed) is the backward link—the displacement of the preceding queue from the present entry. A queue is specified by a queue header, which also consists of two longword links.

The following shows some examples of queue operations. An empty queue is specified by its header at address H. Since the queue is empty, the self-relative links must be 0, as shown below:



If an entry at address B is inserted into an empty queue (at either the head or tail), the queue is as shown below:





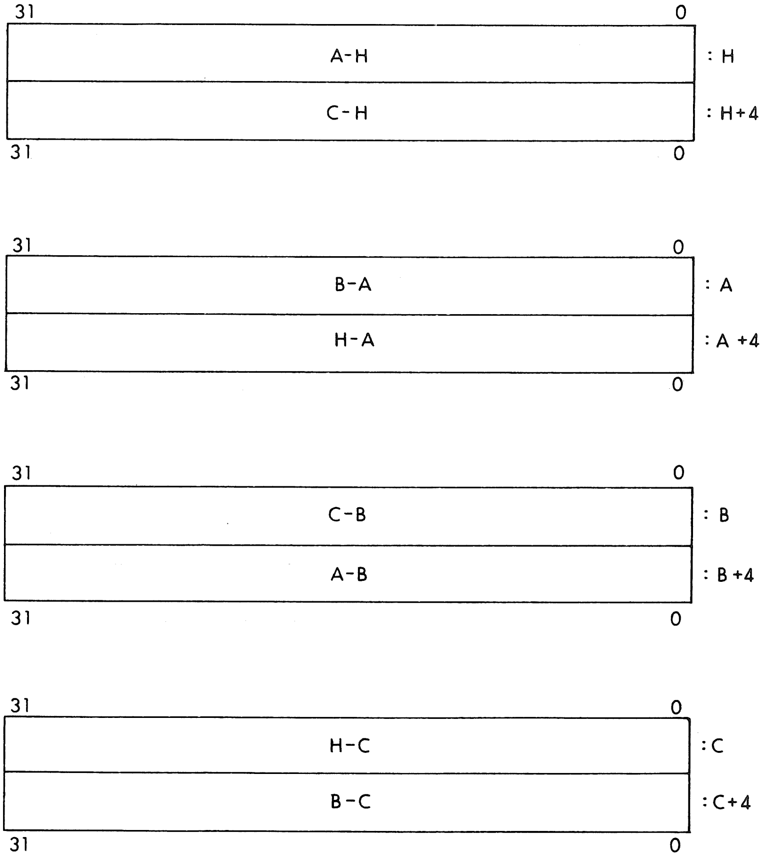
If an entry at address A is inserted at the head of the queue, the queue is as shown below:

31 0	
A-H	: H
B-H	: H +4
31 0	

31 0	
B-A	: A
H-A	: A +4
31 0	

31 0	
H-B	: B
A-B	: B + 4
31 0	

Finally, if an entry at address C is inserted at the tail, the queue appears as follows:



Following the above steps in reverse order yields the effect of removal at the tail and removal at the head.

Four operations can be performed on self-relative queues; insert at and remove from head, insert at and remove from tail. Furthermore, these operations are interlocked to allow cooperating processes in a multiprocessor system to access a shared list without additional synchronization. Queue entries must be quadword aligned. A hardware supported interlocked memory access mechanism is used to read the queue header. Bit 0 of the queue header is used as a secondary interlock and is set when the queue is being accessed. If an interlocked queue instruction encounters the secondary interlock set, it terminates after setting the condition codes to indicate failure to gain access to the queue. If the secondary interlock bit is not set, then the interlocked queue instruction sets it during its operation and clears it at instruction completion. This prevents other interlocked queue instructions from operating on the same queue.

INSQHI

INSERT ENTRY INTO QUEUE AT HEAD, INTERLOCKED

Purpose: interlocked entry insert at head of queue

Format: opcode entry.ab, header.aq

Operation:

```
tmp1 ← (header){interlocked};          !acquire hardware
                                       !interlock
                                       !must have write
                                       !access to header
                                       !header must be
                                       !quadword aligned
                                       !header cannot be
                                       !equal to entry
                                       !tmp1<2:1> must
                                       !be zero

if tmp1<0> EQLU 1 then
(header){interlocked}←tmp1;            !release hardware
                                       !interlock
{set condition codes and terminate instruction};
end;
else
begin
(header){interlocked}←tmp1 v 1         !set secondary
                                       !interlock
                                       !release hardware
                                       !interlock
If {all memory accesses can be completed} then
              !check if following
              !addresses can be written
              !without causing a memory
              !management exception:
              !        entry
              !        header + tmp1
              !Also, check for quadword alignment
begin
              {insert entry into queue};
              {release secondary interlock};
end;
else
begin
              {release secondary interlock};
              {backup instruction};
              {initiate fault};
                 end;
end;
```

Condition Codes:

```
if {insertion succeeded} then
  begin
  N ← 0;
  Z ← (entry) EQL (entry+4);          !first entry in
                                      !queue;
  V ← 0;
  C ← 0;
  end;
else
  begin
  N ← 0;
  Z ← 0;
  V ← 0;
  C ← 1                               !secondary interlock
                                      !failed;
  end
```

Exceptions: Reserved operand

Opcodes: 5C INSQHI Insert Entry into Queue at Head, Interlocked

Description: The entry specified by the entry operand is inserted into the queue following the header. If the entry inserted was the first one in the queue, the conditon code Z-bit is set; otherwise it is cleared. The insertion is a noninterruptible operation. The insertion is interlocked to prevent concurrent interlocked insertions or removals at the head or tail of the same queue by another process even in a multiprocessor environment. Before performing any part of the operation, the processor validates that the entire operation can be completed. This ensures that if a memory management exception occurs, the queue is left in a consistent state. If the instruction fails to acquire the seconary interlock, the instruction sets condition codes and terminates.

Notes:

1. Because the insertion is noninterruptible, processes running in kernel mode can share queues with interrupt service routines.
2. The INSQHI instruction is implemented such that cooperating software processes in a multiprocessor may access a shared list without additional synchronization.
3. To set a software interlock realized with a queue, the following can be used:

```
INSERT:  INSQHI ...          ;was queue empty?
         BEQL     1$         ;yes
         BCS      INSERT     ;try inserting again
         CALL     WAIT(...)  ;no, wait
1$:
```

4. During access validation, any access which cannot be completed results in a memory management exception even though the queue insertion is not started.

5. A reserved operand fault occurs if entry or header is an address that is not quadword aligned (i.e., <2:0>NEQU 0) or if (header)<2:1> is not zero. A reserved operand fault also occurs if header equals entry. In this case the queue is not altered

INSQTI

INSERT ENTRY INTO QUEUE AT TAIL, INTERLOCKED

Purpose: interlocked entry insert at tail of queue

Format: opcode entry.ab, header.aq

Operation: Same as INSQHI, except that the line which reads:

```
        !         header + tmp1
        is replaced by
        !         header + (header + 4)
```

Condition Codes:

```
if {insertion succeeded} then
   begin
   N ← 0;
   Z ← (entry) EQL (entry+4)          !first entry
                                      !in queue;
   V ← 0;
   C ← 0;
   end;
else
   begin
   N ← 0;
   Z ← 0;
   V ← 0;
   C ← 1;                             !secondary interlock
failed
   end
```

Exceptions: Reserved operand

Opcodes:

5D	INSQTI	Insert Entry into Queue at Tail, Interlocked

Description: The entry specified by the entry operand is inserted into the queue preceding the header. The rest of the description is identical to that of INSQHI, immediately above.

Notes:

1. Because the insertion is noninterruptible, processes running in kernel mode can share queues with interrupt service routines.
2. The INSQTI instruction is implemented such that cooperating software processes in a multiprocessor may access a shared list without addtional synchronization.
3. To set a software interlock realized with a queue, see Note 3 at INSQHI, above.
4. During access validation, any access which cannot be completed results in a memory management exception even though the queue insertion was not started.

5. A reserved operand fault occurs if entry, header, or (header+4) is an address that is not quadword aligned (i.e., <2:0> NEQU 0) or if (header)<2:1> is not zero. A reserved operand fault also occurs if header equals entry. In this case the queue is not altered.

REMQHI

REMOVE ENTRY FROM QUEUE AT HEAD, INTERLOCKED

Purpose: interlocked remove of entry from head of queue.

Format: opcode header.aq, addr.wl

Operation:

```
tmp1 ← (header){interlocked};           !acquire hardware
                                        !interlock
                                        !must have write
                                        !access to header
                                        !header must be
                                        !quadword aligned
                                        !header cannot equal
                                        !address of addr.
                                        !tmp1<2:1> must
                                        !be zero
if tmp1<0> EQLU 1 then
begin
(header){interlocked} ← tmp1;           !release hardware
                                        !interlock
{set condition codes and terminate instruction};
end;
else
begin
(header){interlocked}←tmp1 v 1          !set secondary
                                                  !interlock
                                        !release hardware in-
terlock
If {all memory accesses can be completed} then
                  !check if following can be done
                  !without causing a memory
                  !management exception:
                  !write addr operand
                  !read content of header + tmp1{if tmp1
                  !NEQU0}
                  !write into header + tmp1 + (header + tmp1)
{if
                  !tmp1 NEQU0}
                  !Also, check for quadword alignment
begin
                  {remove entry from queue};
                  {release secondary interlock};
end;
else
                  begin
                  {release secondary interlock};
```

```
                {backup instruction};
                {initiate fault};
                   end;
    end:
```

Condition Codes:

```
if {removal succeeded} then
   begin
   N <- 0;
   Z <- (header) EQL 0;                      !queue empty
   V <- tmp1 EQL 0;                          !no entry to remove
   C <- 0;
   end;
else
   begin
   N <- 0;
   Z <- 0;
   V <- 1;               !did not remove anything
   C <- 1;               !secondary interlock failed
   end
```

Exceptions: Reserved operand

Opcodes: 5E REMQHI
Remove Entry from Queue at

Tail, Interlocked

Description: The queue entry following the header is removed from the queue. The address operand is replaced by the address of the entry removed. If no entry was removed from the queue (because either there is nothing to remove or secondary interlock failed), the condition code V bit is set; otherwise it is cleared. If the interlock succeeded and the queue is empty at the end of this instruction, the conditon code Z-bit is set; otherwise it is cleared. The removal is interlocked to prevent concurrent interlocked insertions or removals at the head or tail of the same queue by another process even in a multiprocessor environment. The removal is a noninterruptible operation. Before performing any part of the operation, the processor validates that the entire operation can be completed. This ensures that if a memory management exception occurs, the queue is left in a consistent state. If the instruction fails to acquire the secondary interlock, the instruction sets condition codes and terminates without altering the queue.

Notes:

1. Because the removal is noninterruptible, processes running in kernel mode can share queues with interrupt service routines.

2. The REMQHI instruction is implemented such that cooperating software processes in a multiprocessor may access a shared list without additional synchronization.

3. To release a software interlock realized with a queue, the following can be used:

```
1$:  REMQHI ...      ;removed last?
    BEQL 2$          ;yes
    BCS 1$           ;try removing again
    CALL ACTIVATE(...)
;Activate other waiters
2$:
```

4. To remove entries until the queue is empty, the following can be used:

```
1$: REMQHI ...       ;anything removed?
         BVS 2$      ;no
          .
         process removed entry
          .
         BR 1$       ;
          .
2$: BCS 1$           ;try removing again
         queue empty
```

5. During access validation, any access which cannot be completed results in a memory management exception even through the queue removal is not started.

6. A reserved operand fault occurs if header or (header + (header)) is an address that is not quadword aligned (i.e. <2:0> NEQU 0) or if (header)<2:1> is not zero. A reserved operand fault also occurs if header equals addr. In this case the queue is not altered.

REMQTI

REMOVE ENTRY FROM QUEUE AT TAIL, INTERLOCKED

Purpose: interlocked entry remove from tail of queue

Format: opcode header.aq, addr.wl

Operation:

```
tmp1 <- (header)
{interlocked};            !acquire hardware
                          !interlock
                          !must have write access to header
                          !header must be quadword aligned
                          !header cannot equal address
                          !of addr
                          !tmp1<2:1> must be zero

if tmp1<0> EQLU 1 then
   begin
   (header){interlocked} <- tmp1;
!release hardware
                          !interlock
   {set condition codes and terminate instruction}:
   end;

else

   begin
   (header) {interlocked} <- tmp1 v 1;
!set secondary
                          !interlock
                          !release hardware interlock
   If {all memory accesses can be completed} then
                          !check if the following can be
                          !done without
                          !causing a memory management
                          !exception:
                          !write addr operand
                          !read contents of header + (header
                          !+ 4) {if tmp1 NEQU 0}
                          !
                          !write into header + (header +4)
                          !         + (header + 4 + (header
                          !+ 4)) {if tmp1 NEQU 0}
                          !Also, check for quadword
                          !alignment
begin
{remove entry from queue};
{release secondary interlock};
                          end;
else
```

```
                         begin
                         {release secondary interlock};
                         {backup instruction};
                         {initiate fault};
                         end;
end;
```

Condition Codes:

```
if {removal succeeded} then
  begin
  N <- 0;
  Z <- (header + 4) EQL 0!queue empty;
  V <- tmp3 EQL 0        !no entry to remove;
  C <- 0;
  end;
else
  begin
  N <- 0;
  Z <- 0;
  V <- 1                 !did not remove anything;
  C <- 1                            !secondary interlock failed;
  end
```

Exceptions: Reserved operand

Opcodes: 5F REMQTI

Remove Entry from Queue at

Tail, Interlocked

Description: The queue entry preceding the header is removed from the queue. The rest of the description matches that of REMQHI, immediately above.

Notes:

1. Because the removal is noninterruptible, processes running in kernel mode can share queues with interrupt service routines.
2. The REMQTI instruction is implemented such that cooperating software processes in a multiprocessor may access a shared list without additional synchronization.
3. To release a software interlock realized with a queue, see note 3, for REMQHI, immediately above.
4. To remove entries until the queue is empty, see note 4, for REMQHI, immediately above.
5. During access validation, any access which cannot be completed results in a memory management exception even though the queue removal is not started.
6. A reserved operand fault occurs if header, (header + 4), or (header + (header + 4)+4) is an address that is not quadword aligned (i.e., <2:0> NEQU 0) or if (header)<2:1> is not zero. A reserved operand fault occurs if header equals addr. In this case the queue is not altered.

VARIABLE LENGTH BIT FIELD INSTRUCTIONS

Variable length bit field instructions are useful when you are dealing with data not in 8-bit increments (for example, 13 bits of data that do not start on a byte boundary). Such data could also be handled without these instructions, but less efficiently, since it would require additional shift and mask operations to get the bits into the proper form and to eliminate the nonrequired bits.

A variable bit field is 0 to 32 contiguous bits (contained in 1 to 5 bytes) that is arbitrarily located with respect to byte boundaries.

The variable length bit field instructions have four operand specifiers; three of these specifiers determine how to find the variable length field and the fourth designates where it is to be stored. The first three specifiers are:

Position Operand—a signed longword operand that designates the number of bits away from the base address operand.

If the variable length field is contained in a register, the position operand must have a value in the range 0 through 31 (if the size is nonzero) or a reserved operand fault occurs.

Size Operand—a byte operand which specifies the length of the field. This operand must be in the range 0 through 32 or a reserved operand fault occurs. The size operand will normally be a short literal if the field is fixed.

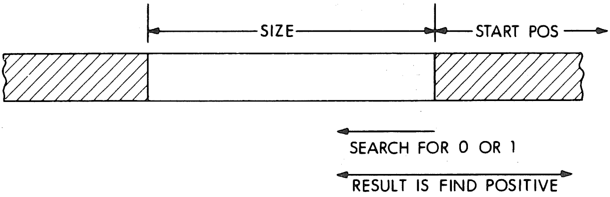
Base Address—an address relative to which the position is used to locate the bit field. The base address is obtained from an "address access" type operand. Unlike other "address access" type operands, register mode may be designated in the specifier. In this case, the field is contained in register n designated by the operand specifier (or R(n+1) concatenated with R(n)).

FIND FIRST

Purpose: locate first bit in bit field

Format: opcode startpos.rl, size.rb, base.vb, findpos.wl

Operation:



Condition Codes:
N ← 0;
Z ← {bit not found};
V ← 0;
C ← 0

Exceptions: Reserved operand

Opcodes:

EB	FFC	Find First Clear
EA	FFS	Find First Set

Description: A field specified by the start position, size, and base operands is extracted. The field is tested for a bit in the state indicated by the instruction starting at bit 0 and extending to the highest bit in the field. If a bit in the indicated state is found, the find position operand is replaced by the position of the bit and the Z condition code bit is cleared. If no bit in the indicated state is found, the find position operand is replaced by the position (relative to the base) of a bit one position to the left of the specified field and the Z condition code bit is set. If the size operand is 0, the find position operand is replaced by the start position operand and the Z condition code bit is set.

Notes:

1. A reserved operand fault occurs if:
   a. size GTRU 32
   b. startpos GTRU 31, size NEQ 0, and the field is contained in the registers
2. On a reserved operand fault, the find position operand is unaffected and the condition codes are unpredictable.

Example:

```
FIND FIRST SET
FFS #5, #20, Work, R3                ;Find first bit
                                     ;set in Work
```

Initial Conditions:
Work = ↑X 00040000 (Bit 18 set)
R3 = 00000000

After Instruction Execution:
Work ↑X 00040000
R3 = 00000012hex (18 decimal)

Example:

```
FIND FIRST CLEAR
FFC #5, #10, Work1, R2           ;Find first clear bit
                                 ;in Work1
```

Initial Conditions:
Work1 = ↑XF00
R2 = 00000000

After Instruction Execution:
Work1 = ↑XF00
R2 = 00000008

Example: When referencing memory, the startpos field may be greater than 31. This provides an effective technique to search an entire array for the first bit set (or clear)

```
   CLRL R0                       ;start at bit 0
10$:FFS R0,#32,ARRAY,R0          ;R0 is incremented
                                 ;by 32 for each
                                 longword searched
   BEQL10#                       ;search next
                                 ;longword
   {R0 is match bit index}
```

The Find First instruction is thus useful when it is desired to search for the first 1 or the first 0 in a string of bits. For example, the operating system might contain a table where each bit represents a block of data on a disk. If the bit is a 1, it indicates that block of data is in use; if the bit is a 0, it indicates the block is free. Consequently, if it is desired to find the first free block, the user would issue a Find First Clear instruction which searches for the first 0 bit in the table.

EXTRACT FIELD

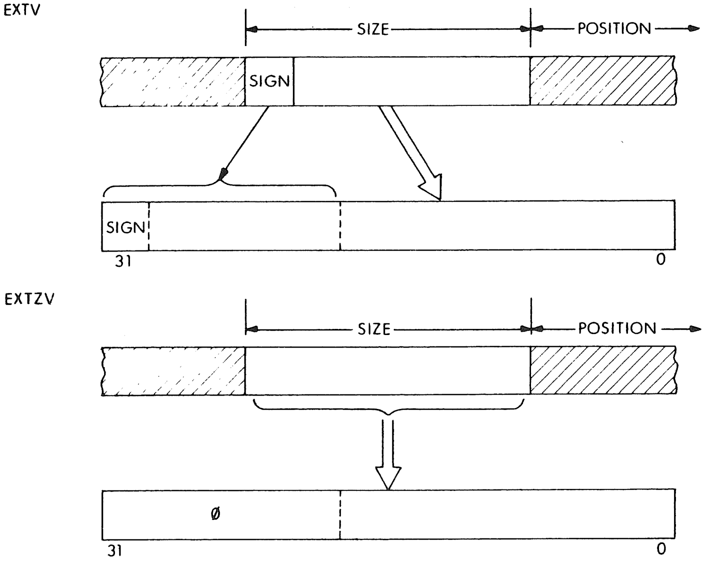
Purpose: moves bit field to integer

Format: opcode pos.rl, size.rb, base.vb, dst.wl

Operation:

```
EXTV:
dst ← if size NEQU 0 then SEXT(FIELD(pos, size, base))
          else 0

EXTZV:
dst ← if size NEQU 0 the ZEXT(FIELD(pos, size, base))
          else 0
```



Condition Codes:

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

Exceptions: Reserved operand

Opcodes:

EE	EXTV	Extract Field
EF	EXTZV	Extract Zero-Extended Field

Description: For EXTV, the destination operand is replaced by the sign-extended field specified by the position, size, and base operands. For EXTZV, the destination operand is replaced by the zero-extended field specified by the position, size, and base operands. If the size operand is 0, the only action is to replace the destination operand with 0 and affect the condition codes.

An example of this instruction is to extract the four protection bits (bits 27 through 30) from the memory management unit Page Table Entry. The base address is the address of a longword operand containing these bits; the position operand could be the number of bits from the base address to the protection code; and the size operand would be 4 since the protection code is 4 bits long. The destination operand would specify where the protection bits are to be stored.

Since the protection code is not an arithmetic operand and does not need to be sign-extended, the Extract Zero-Extended Field instruction should be specified.

Notes:

1. A reserved operand fault occurs if:
   a. size GTRU 32
   b. pos GTRU 31, size NEQ 0, and the field is contained in the registers.
2. On a reserved operand fault, the destination operand is unaffected and the condition codes are unpredictable.

Example:

```
EXTRACT FIELD
EXTV #5, #10, Work1, R0  ;put bits 5 thru 14
                         ;from Work1 into R0
```

Initial Conditions:
Work1 = 00004F04
R0 = 00000000

After Instruction Execution:
Work1 = 00004F04
R0 = FFFFFC78

Example:

```
EXTRACT FIELD, ZERO EXTENDED
EXTZV #5, #10, Work1, R1          ;put bits 5 thru 15
                                  ;from Work1 into R1
                                  ;and clear bits 11
                                  thru 31
```

Initial Conditions:
Work1 = 00004F04
R1 =00000000

After Instruction Execution:
Work1 = 00004F04
R1 = 00000478

COMPARE FIELD

Purpose: compare bit field to integer

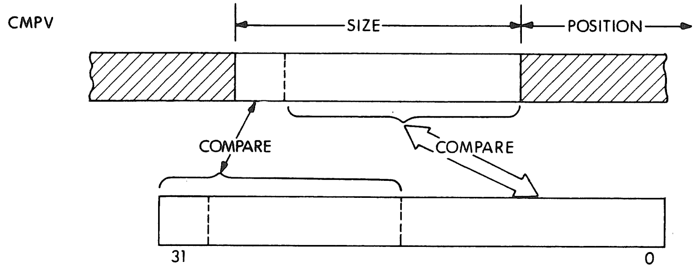
Format: opcode pos.rl, size.rb, base.vb, src.rl

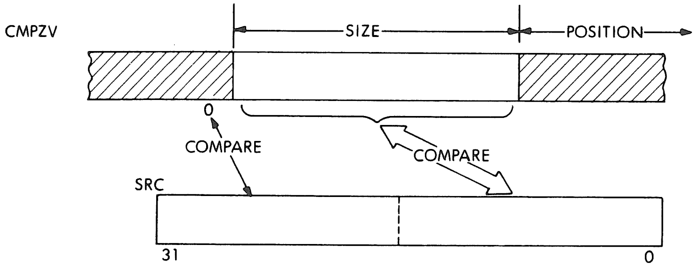
Operation:

```
CMPV:
tmp ← if size NEQU 0 then SEXT(FIELD (pos, size, base)) else
0;

CMPZV:
tmp ← if size NEQU 0 then ZEXT(FIELD (pos, size, base)) else
0:

tmp − src;
```





Condition Codes:

N ← tmp LSS src;
Z ← tmp EQL src;
V ← 0;
C ← tmp LSSU src

Exceptions: Reserved operand

Opcodes:

EC	CMPV	Compare Field
ED	CMPZV	Compare Zero-Extended Field

Description: The field specified by the position, size, and base operands is compared with the source operand. For CMPV, the source operand is compared with the sign-extended field. For CMPZV, the source operand is compared with the zero-extended field. The only action is to affect the condition codes.

Notes:

1. A reserved operand fault occurs if:
   a. size GTRU 32
   b. pos GTRU 31, size NEQ 0 and the field is contained in the registers.
2. On a reserved operand fault, the condition codes are unpredictable.

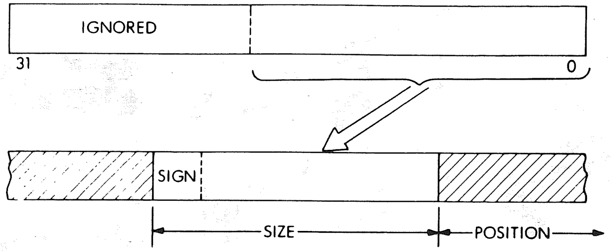
INSV

INSERT FIELD

Purpose: move integer to bit field

Format: opcode src.rl, pos.rl, size.rb, base.vb

Operation: if size NEQU then FIELD(pos, size, base) ← src <{size −1}>;



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

Exceptions: Reserved operand

Opcodes: F0 INSV Insert Field

Description: The field specified by the position, size, and base operands is replaced by bits size 1:0 of the source operand. If the size operand is 0, the only action is to affect the condition codes.

Notes:

1. A reserved operand fault occurs if:
   a. size GTRU 32
   b. pos GTRU 31, size NEQ 0, and the field, is contained in the registers.
2. On a reserved operand fault, the field is unaffected and the condition codes are unpredictable.

Example:

```
INSERT FIELD
INSV R0, #16, #10, Work   ;put bits 0 thru 9
                          ;of R0 into bits 16 thru
                          ;25 of Work
```

Initial Conditions:
Work = FFFFFFFF
R0 = 00000078

After Instruction Execution:
Work = FC78FFFF
R0 = 00000078

digital VAX 11/750

CHAPTER 13
CONTROL INSTRUCTIONS

This chapter describes the branch, loop, control, subroutine, case, and call classes of instructions. In most implementations of the VAX architecture, improved execution speed will result if the target of a control instruction is on an aligned longword boundary, but this is not a requirement.

BRANCH AND JUMP INSTRUCTIONS

The two basic types of control transfer instructions are **Branch** and **Jump** instructions. Both Branch and Jump load new addresses in the Program Counter. With Branch instructions, you supply a displacement (offset) which is added to the current contents of the Program Counter to obtain the new address, while with Jump instructions, you supply the address you want loaded, using one of the normal addressing modes.

Because most transfers are to locations relatively close to the current instructions, and because Branch instructions take less space than Jump instructions, the processor offers a variety of Branch instructions to choose from. There are two unconditional Branch instructions and many Conditional Branch instructions, such as Branch on Less Than and Branch on Less Than Unsigned.

The Unconditional Branch instructions allow you to specify a byte-size (BRB) or word-size displacement (BRW), which means you can branch to locations as far from the current location as 32,767 (i.e., ½ $(2^{16} - 1)$) bytes in either direction. For control transfers to locations farther away, use the Jump instruction (JMP).

Two special types of Branch and Jump instruction are provided for calling subroutines: Branch to Subroutine (BSB) and Jump to Subroutine (JSB). Both BSB and JSB instructions save the contents of the Program Counter on the stack before loading the Program Counter with the new address. With Branch to Subroutine, you can supply either a byte (BSBB) or word (BSBW) displacement.

This shortcut to subroutine calling is complemented by the Return from Subroutine (RSB) instruction. RSB pops the first longword off the stack and loads it into the Program Counter. Since the Branch to Subroutine instruction is either two or three bytes long, and the Return from Subroutine instruction is one byte long, it is possible to write extremely efficient programs using subroutines.

Dispatching to a routine based on the value of a variable occurs frequently enough that some high-level languages include special constructs to handle it, such as the computed GO TO in FORTRAN and the Case statement in PASCAL. Because of this, the VAX instruction set includes a Case instruction so that such control structures can be represented efficiently. Not only does Case handle the transfer of control, but it also handles the initialization and bounds checking for the INDEX variable.

The objective of the Case statement is to transfer control to one of *n* locations based on the value of the integer selector operand. The base operand specifies the lower bound for selector. Following the Case instruction is a table of word displacements for the *n* branch locations. Just as the displacements in branch instructions are added to the PC to give the branch destination, these word displacements are added to the address of the first displacement to form the Case branch destinations.

BRANCH ON (CONDITION)

Purpose: test condition code

Format: opcode displ.bb

Operation: if condition then PC ← PC + SEXT (displ);

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

Exceptions: None

Opcodes:

CONDITION		
12 Z EQL 0	BNEQ	Branch on Not Equal (signed)
	BNEQU	Branch on Not Equal Unsigned
13 Z EQL 1	BEQL	Branch on Equal (signed)
	BEQLU	Branch on Equal Unsigned
14 {N OR Z}EQL 0	BGTR	Branch on Greater Than (signed)
15 {N OR Z}EQL 1	BLEQ	Branch on Less Than or Equal (signed)
18 N EQL 0	BGEQ	Branch on Greater Than or Equal (signed)
19 N EQL 1	BLSS	Branch on Less Than (signed)
1A {C OR Z} EQL 0	BGTRU	Branch on Greater Than Unsigned
1B {C or Z} EQL 1	BLEQU	Branch Less Than or Equal Unsigned
1C V EQL 0	BVC	Branch on Overflow Clear
1D V EQL 1	BVS	Branch on Overflow Set
1E C EQL 0	BGEQU	Branch on Greater Than or Equal Unsigned
	BCC	Branch on Carry Clear
1F C EQL 1	BLSSU	Branch on Less Than Unsigned
	BCS	Branch on Carry Set

Description: The condition codes are tested, and if the condition indicated by the instruction is met, the sign-extended branch displacement is added to the PC and PC is replaced by the result.

Notes:

The VAX conditional branch instructions permit considerable flexibility in branching but you need to exercise some care to choose the correct one. The conditional branch instructions are divided into 3 overlapping groups:

1. The Overflow and Carry Group

BVS	V EQL 1
BVC	V EQL 0
BCS	C EQL 1
BCC	C EQL 0

These instructions are typically used to check for overflow (when overflow traps are not enabled), for multiprecision arithmetic, and for other special purposes.

2. The Unsigned Group

BLSSU	C EQL 1
BLEQU	{C or Z} EQL 1
BEQLU	Z EQL 1
BNEQU	Z EQL 0
BGEQU	C EQL 0
BGTRU	{C OR Z} EQL 0

These instructions typically follow integer and field instructions where the operands are treated as unsigned integers, addressed instructions, and character string instructions.

3. The Signed Group

BLSS	N EQL 1
BLEQ	{N OR Z} EQL 1
BEQL	Z EQL 1
BNEQ	Z EQL 0
BGEQ	N EQL 0
BGTR	{N OR Z} EQL 0

These instructions typically follow integer and field instructions where the operands are being treated as signed integers, floating point instructions, and decimal string instructions.

BR
JMP

BRANCH, JUMP

Purpose: transfer control

Format:
opcode displ.bx !Branch
opcode dst.ab !Jump

Operation:
PC ← PC +SEXT (displ); !Branch
PC ← dst; !Jump

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

Exceptions: None

Opcodes:

11	BRB	Branch With Byte Displacement
31	BRW	Branch With Word Displacement
17	JMP	Jump

Description: For Branch, the sign-extended branch displacement is added to PC and PC is replaced by the result. For Jump, the PC is replaced by the destination operand.

BB

BRANCH ON BIT

Purpose: test selected bit

Format: opcode pos.rl, base.vb, displ.bb

Operation:

```
teststate = if {BBS} then 1 else 0;
if FIELD (pos, 1, base) EQL teststate then
PC ← PC + SEXT (displ);
```

Condition Codes:

```
N ← N;
Z ← Z;
V ← V;
C ← C
```

Exceptions: Reserved operand

Opcodes:

E0	BBS	Branch on Bit Set
E1	BBC	Branch on Bit Clear

Description: The single bit field specified by the position and base operands is tested. If it is in the test state indicated by the instruction, the sign-extended branch displacement is added to PC and PC is replaced by the result.

Notes:

1. A reserved operand fault occurs if pos GTRU 31 and the bit is contained in a register.
2. On a reserved operand fault, the condition codes are unpredictable.

BRANCH ON BIT (AND MODIFY WITHOUT INTERLOCKED)

Purpose: test and modify selected bit

Format: opcode pos.rl, base.vb, displ.bb

Operation:

```
teststate = if {BBSS or BBSC} then 1 else 0;
newstate = if {BBSS or BBCS} then 1 else 0;
temp ← FIELD (pos, 1, base);
FIELD (pos, 1, base) ← newstate;
if tmp EQL teststate then
PC ← PC + SEXT (displ);
```

Condition Codes:

N ← N:
Z ← Z;
V ← V;
C ← C

Exceptions: Reserved operand

Opcodes:

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

Description: The single bit field specified by the position and base operands is tested. If it is in the test state indicated by the instruction, the sign-extended branch displacement is added to PC and PC is replaced by the result. Regardless of whether the branch is taken or not, the tested bit is put in the new state as indicated by the instruction.

Notes:

1. A reserved operand fault occurs if pos GTRU 31 and the bit is contained in a register.
2. On a reserved operand fault, the field is unaffected and the condition codes are unpredictable.
3. The modification of the bit is not an interlocked operation. See BBSSI and BBCCI for interlocking instructions.

BRANCH ON BIT INTERLOCKED

Purpose: test and modify selected bit under memory interlock

Format: opcode pos.rl, base.vb, displ.bb

Operation:

```
teststate = if {BBSSI} then 1 else 0;
newstate = teststate;
{set interlock};
temp <- FIELD (pos, 1, base);
FIELD (pos, 1, base) <- newstate;
{release interlock};
if tmp EQL teststate then
PC <- PC + SEXT (displ);
```

Condition Codes:

```
N <- N;
Z <- Z;
V <- V;
C <- C
```

Exceptions: Reserved operand

Opcodes:

E6	BBSSI Branch on Bit Set and Set Interlocked
E7	BBCCI Branch on Bit Clear and Clear Interlocked

Description: The single bit field specified by the position and base operands is tested. If it is in the test state indicated by the instruction, the sign-extended branch displacement is added to the PC and PC is replaced by the result. Regardless of whether the branch is effected or not, the tested bit is put in the new state as indicated by the instruction. If the bit is contained in memory, the reading of the state of the bit and the setting of it to the new state constitute an interlocked operation, interlocked against similar operations by other processors or devices in the system.

Notes:

1. A reserved operand fault occurs if pos GTRU 31 and the bit is contained in registers.
2. On a reserved operand fault, the field is unaffected and the condition codes are unpredictable.
3. Except for memory interlocking, BBSSI is equivalent to BBSS and BBCCI is equivalent to BBCC.

Example: This instruction is designed to implement interlock with other processors or devices. For example, to implement "busy waiting":

```
1$:     BBSSI     bit,base,1$
```

BLB

BRANCH ON LOW BIT

Purpose: test bit

Format: opcode src.rl, displ.bb

Operation:

```
teststate = if {BLBS} then 1 else 0;
if src<0> EQL teststate then
PC ← PC + SEXT (displ);
```

Condition Codes:

```
N ← N;
Z ← Z;
V ← V;
C ← C
```

Exceptions: None

Opcodes:

E8	BLBS	Branch on Low Bit Set
E9	BLBC	Branch on Low Bit Clear

Description: The low bit (bit 0) of the source operand is tested and if it is equal to the test state indicated by the instruction, the sign-extended branch displacement is added to PC and PC is replaced by the result.

Notes: The source operand is taken with longword context although only one bit is tested.

Example:

```
                BLBC        R0, ERROR
                 .
                 .
                 .
                 .
                 .
ERROR:           .
                 .
```

IF the low bit of the contents of R0 is 0, then the program branches to ERROR. This could be used as a parity check.

LOOP CONTROL INSTRUCTIONS

ACB

ADD COMPARE AND BRANCH

Purpose: maintain loop count and loop

Format: opcode limit.rx, add.rx, index.mx, displ.bw

Operation:

```
index ← index + add;
if { {add GEQ 0} AND {index LEQ limit} } OR
{ {add LSS 0} AND {index GEQ limit} } then
PC ← PC + SEXT (displ);
```

Condition Codes:

N ← index LSS 0;
Z ← index EQL 0;
V ← {integer or floating overflow};
C ← C

Exceptions:

Integer overflow
Floating overflow
Floating underflow
Reserved operand

Opcodes:

9D	ACBB	Add Compare and Branch Byte
3D	ACBW	Add Compare and Branch Word
F1	ACBL	Add Compare and Branch Longword
4F	ACBF	Add Compare and Branch F_floating
6F	ACBD	Add Compare and Branch D_floating
4FFD	ACBG	Add Compare and Branch G_floating
6FFD	ACBH	Add Compare and Branch H_floating

Description: The addend operand is added to the index operand and the index operand is replaced by the result. The index operand is compared with the limit operand. If the addend operand is positive (or 0) and the comparison is less than or equal, or if the addend is negative and the comparison is greater than or equal, then the sign-extended branch displacement is added to PC and PC is replaced by the result.

Notes:

1. ACB efficiently implements the general FOR or DO loops in high-level languages, since the sense of the comparison between index and limit is dependent on the sign of the addend.

2. On integer overflow, the index operand is replaced by the low order bits of the true result. Comparison and branch determination proceed normally on the updated index operand.
3. On floating underflow and floating overlow, the actions taken depend upon the VAX processor being used.
4. On a reserved operand fault, the index operand is unaffected and the condition codes are unpredictable.
5. Except for 4 above, the C-bit is unaffected.

AOB

ADD ONE AND BRANCH

Purpose: increment integer loop count and loop

Format: opcode limit.rl, index.ml, displ.bb

Operation:

```
index ← index +1;
if index LSS limit                          !AOBLSS
then PC ← PC + SEXT (displ);
if index LEQ limit                          !AOBLEQ
then PC ← PC + SEXT (displ);
```

Condition Codes:

```
N ← index LSS 0;
Z ← index EQL 0;
V ← {integer overflow};
C ← C
```

Exceptions: Integer overflow

Opcodes:

F2	AOBLSS	Add One and Branch Less Than
F3	AOBLEQ	Add One and Branch Less Than or Equal

Description: One is added to the index operand and the index operand is replaced by the result. The index operand is compared with the limit operand. On AOBLSS, if it is less than, the branch is taken. On AOBLEQ, if it is less than or equal, the branch is taken. If the branch is taken, the sign-extended branch displacement is added to the PC and the PC is replaced by the result.

Notes:

1. Integer overflow occurs if the index operand before addition is the largest positive integer. On overflow, the index operand is replaced by the largest negative integer, and thus (unless the limit operand is the largest negative integer on AOBLSS) the branch is taken.
2. The C-bit is unaffected.

SOB

SUBTRACT ONE AND BRANCH

Purpose: decrement integer loop count and loop

Format: opcode index.ml, displ.bb

Operation:

```
index ← index −1;
If index GEQ 0 then                    !SOBGEQ
PC ← PC + SEXT (displ);
index ← index -1;
If index GTR 0 then                    !SOBGTR
PC ← PC + SEXT (displ);
```

Condition Codes:

```
N ← index LSS 0;
Z ← index EQL 0;
V ← {integer overflow};
C ← C
```

Exceptions: Integer overflow

Opcodes:

F4	SOBGEQ	Subtract One and Branch Greater Than or Equal
F5	SOBGTR	Subtract One and Branch Greater Than

Description: One is subtracted from the index operand and the index operand is replaced by the result. On SOBGEQ, if the index operand is greater than or equal to 0, the branch is taken. On SOBGTR, if the index operand is greater than 0, the branch is taken. If the branch is taken, the sign-extended branch displacement is added to the PC and the PC is replaced by the result.

Notes:

1. Integer overflow occurs if the index operand before subtraction is the largest negative integer. On overflow, the index operand is replaced by the largest positive integer, and thus the branch is taken.
2. The C-bit is unaffected.

Example:

```
                MOVL                                        #1, R1
                MOVL                                        #5, R0
LOOP:
                MULL2                                       R0, R1
                SOBGTR                                      R0,
LOOP
                    .
                    .
                    .
```

The procedure implements 5! calculation. The result of the first multiplication (5) is placed in R1; R0 is reduced to 4; the next multiplication yields places 20 in R1; and so on. Procedure continues as long as the value in R0 is greater than or equal to 0.

CASE

Purpose: perform multiway branching depending on arithmetic input

Format: opcode selector.rx, base.rx, limit.rx, displ[0].bw,...,displ[limit].bw

Operation:

```
tmp ← selector – base;
PC ← PC + if tmp LEQU limit then
SEXT (displ [tmp]) else {2 + 2*SEXT (limit)};
```

Condition Codes:

```
N ← tmp LSS limit;
Z ← tmp EQL limit;
V ← 0;
C ← tmp LSSU limit
```

Exceptions: None

Opcodes:

8F	CASEB	Case Byte
AF	CASEW	Case Word
CF	CASEL	Case Longword

Description: The base operand is subtracted from the selector operand and a temporary is replaced by the result. The temporary is compared with the limit operand and if it is less than or equal unsigned, a branch displacement selected by the temporary value is added to PC and PC is replaced by the result. Otherwise, 2 times the sum of the limit operand plus 1 is added to PC and PC is replaced by the result. This causes PC to be moved past the array of branch displacements. Regardless of the branch taken, the condition codes are affected by the comparison of the temporary operand with the limit operand.

Notes:

1. After operand evaluation, PC is pointing at displ[0], not the next instruction. The branch displacements are relative to the address of displ[0].
2. The selector and base operands can both be considered either as signed or unsigned integers.

Example: This instruction implements high-level language computed GO TO statements. You supply a list of displacements that generate different branch addresses depending on the value you obtain as a selector. The branch falls through if the selector does not generate any of the displacements on the list.

The FORTRAN STATEMENT
  GO TO (10, 20, 30), I
is equivalent to

```
      CASEL I, #1, #3        ;only values 1,2,3 are valid
1$    .WORD 10$-1$, #3       ;if 1
      .WORD 20$-1$           ;if 2
      .WORD 20$-1$           ;if 3
                             ;fall through if out of range
          .
          .
          .
10$:
          .
          .
          .
20$:
          .
          .
          .
30$:
          .
          .
          .
```

BSB
JSB

BRANCH TO SUBROUTINE
JUMP TO SUBROUTINE

Purpose: transfer control to subroutine

Format:

```
opcode displ.bx                    !branch to subroutine
opcode dst.ab                      !jump to subroutine
```

Operation:

```
-(SP) ← PC;
PC ← PC + SEXT (displ);!branch to subroutine
PC ← dst                        !jump to subroutine
```

Condition Codes:

N ← N;
Z ← Z;
V ← V;
C ← C

Exceptions: None

Opcodes:

10	BSBB	Branch to Subroutine with Byte Displacement
30	BSBW	Branch to Subroutine with Word Displacement
16	JSB	Jump to Subroutine

Description: PC is pushed on the stack as a longword. For Branch, the sign-extended branch displacement is added to PC and PC is replaced by the result. For Jump, PC is replaced by the destination operand.

Notes: Since the operand specifier conventions cause the evaluation of the destination operand before saving PC, JSB can be used for coroutine calls, with the stack used for linkage. The form of such a call is JSB @(SP)+.

RETURN FROM SUBROUTINE

Purpose: return control from subroutine

Format: opcode

Operation: PC ← (SP)+;

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

Exceptions: None

Opcodes:

05	RSB	Return from Subroutine

Description: PC is replaced by a longword popped from the stack.

Notes:

1. RSB is used to return from subroutines called by the BSBB, BSBW, and JSB instructions.
2. RSB is equivalent to JMP @(SP)+, but is shorter.

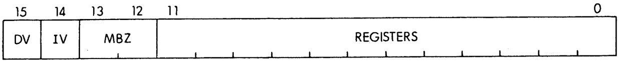
PROCEDURE CALL INSTRUCTIONS

Procedures are general purpose routines that use argument lists passed automatically by the processor and use only local variables for data storage. A procedure call instruction provides several services. It:

- Saves all the registers that the procedure uses, and only those registers, before entering the procedure
- Passes an argument list to a procedure
- Maintains the Stack, Frame, and Argument Pointers
- Sets the arithmetic trap enables to a specific state

Three instructions are used to implement a standard procedure calling interface. Two instructions implement the Call to the procedure; the third implements the matching Return. CALLG calls a procedure with the argument list actuals (i.e., actual arguments rather than formal arguments) in an arbitrary location. The CALLS instruction calls a procedure with the argument list actuals on the stack. Upon return after a CALLS this list is automatically removed from the stack. Both call instructions specify the address of the entry point of the procedure being called. It is assumed to consist of a word termed the **entry mask** followed by the procedure's instructions. The procedure terminates by executing a RET instruction.

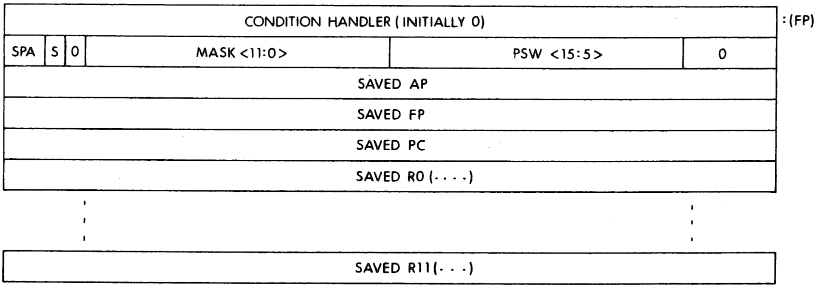
The entry mask specifies the subprocedure's register use and overflow enables:



On Call the stack is aligned to a longword boundary and the trap enables in the PSW are set to a known state to ensure consistent behavior of the called procedure. Integer overflow enable and numeric overflow enable are affected according to bits 14 and 15 of the entry mask respectively; floating underflow enable is cleared.

R11 through R0, specified by bits 11 through 0, respectively, are saved on the stack and are restored by the RET instruction. The procedure calling standard requires that all registers in the range R2 through R11 used in the procedure must appear in the mask. In addition, the Call instructions always preserve PC, SP, FP, and AP. Thus, a procedure can be considered equivalent to a complex instruction which stores a value into R0 and R1, affects memory, and clears the condition codes. If the procedure has no function value, the contents of R0 and R1 are unpredictable.

In order to preserve the state, the Call instructions form a structure on the stack termed a Call Frame or Stack Frame. This contains the saved registers, the saved PSW, the register save mask, and several control bits. The frame also includes a longword which the Call instructions clear; this is used to implement the condition handling facility. (Refer to Appendix C.) At the end of execution of the Call instruction, FP contains the address of the Stack Frame. The RET instruction uses the contents of FP to find the Stack Frame and restore state. The condition handling facility assumes that FP always points to the Stack Frame, which has the following format:



(0 TO 3 BYTES SPECIFIED BY SPA, STACK POINTER ALIGNMENT)
S BIT = SET IF CALLS; CLEAR IF CALLG.

Note that the saved condition codes and the saved trace enable are cleared. The contents of the frame PSW <3:0> at the time RET is executed will become the condition codes resulting from the execution of the procedure.

The software defines symbolic names for the fixed fields in the call frame as follows:

Mnemonic	**Value**	**Meaning**
SF$A_HANDLER	0	condition handler
SF$W_SAVE_PSW	4	saved PSW
SF$W_SAVE_MASK	6	offset, CALLS, and mask
SF$L_SAVE-AP	8	saved AP
SF$L_SAVE_FP	12	saved FP (backward link)
SF$L_SAVE_PC	16	saved PC
SF$L_SAVE_REGS	20	start of saved R0..R11

The savepsw fields have symbolic names as follows:

SF$a_C	<0>	saved C condition code
SF$a_V	<1>	saved V condition code
SF$a_Z	<2>	saved Z condition code
SF$a_N	<3>	saved N condition code
SF$a_TBIT	<4>	saved trace enable
SF$a_IV	<5>	saved integer overflow enable
SF$a_FU	<6>	saved floating underflow enable
SF$a_DV	<7>	saved divide overflow enable

The save_mask fields have these symbolic names:

SF$b_SAVE_MASK	<11:0>	register mask
SF$b_CALLS	<13>	CALLS flag
SF$b_STACKOFFS	<15:14>	stack alignment

a = M,S, or V for mask, size, or position
b = S or V for size or position

These names are defined by the $SFDEF macro in the system library.

CALLG

CALL PROCEDURE WITH GENERAL ARGUMENT LIST

Purpose: invoke a procedure with actual arguments from anywhere in memory

Format: opcode arglist.ab, dst.ab

Operation:

```
{align stack};
{create stack frame};
{set arithmetic trap enables};
{set new values of AP, FP, PC}
```

Condition Codes:

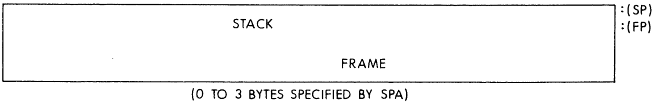
N ← 0;
Z ← 0;
V ← 0;
C ← 0

Exceptions: Reserved operand

Opcodes:

FA	CALLG	Call Procedure with General Argument List

Description: SP is saved in a temporary and then bits 1:0 are replaced by 0 so that the stack is longword aligned. The procedure entry mask is scanned from bit 11 to bit 0 and the contents of registers whose number corresponds to set bits in the mask are pushed on the stack as longwords, along with PC, FP, and AP. The condition codes are cleared. A longword containing the saved two low bits of SP in bits 31:30, a 0 in bit 29 and bit 28, the low 12 bits of the procedure entry mask in bits 27:16, and the PSW in bits 15:0, with T cleared, is pushed on the stack. A longword 0 is pushed on the stack. FP is replaced by SP. AP is replaced by the arglist operand which specifies the address of the actual argument list. The trap enables in the PSW are set to a known state. Integer overflow and decimal overflow are affected according to bits 14 and 15 of the entry mask respectively; floating underflow is cleared. T-bit is unaffected. PC is replaced by the sum of destination operand and 2, which transfers control to the called procedure at the byte beyond the entry mask.



Notes:

1. If bits 13:12 of the entry mask are not 0, a reserved operand fault occurs.
2. On a reserved operand fault, conditon codes are unpredictable.
3. The procedure calling standard and the condition handling facility require the following register saving conventions. R0 and R1 are always available for function return values and are never saved in the entry mask. All registers 2 through 11 which are modified in the called procedure must be preserved in the mask. (Refer to Appendix C.)

CALLS

CALL PROCEDURE WITH STACK ARGUMENT LIST

Purpose: invoke a procedure with actual arguments or addresses on the stack

Format: opcode numarg.rl,dst.ab

Operation:

```
{push arg count};
{align stack};
{create stack frame};
{set arithmetic trap enables};
{set new values of AP, FP, PC}
```

Condition Codes:

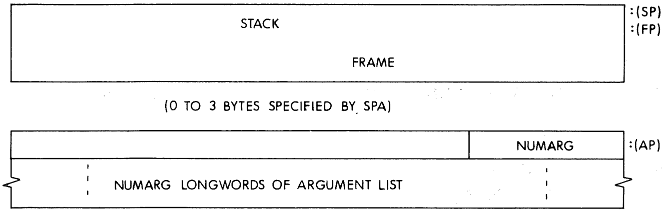
N ← 0;
Z ← 0;
V ← 0;
C ← 0

Exceptions: Reserved operand

Opcodes:

FB	CALLS	Call Procedure With Stack Argument List

Description: The numarg operand is pushed on the stack as a longword (byte 0 contains the number of arguments; the high-order 24 bits are used by DIGITAL software). SP is saved in a temporary and then bits 1:0 of SP are replaced by 0 so that the stack is longword aligned. The procedure entry mask is scanned from bit 11 to bit 0 and the contents of register whose number corresponds to set bits in the mask are pushed on the stack. PC, FP, and AP are pushed on the stack as longwords. The condition codes are cleared. A longword containing the saved two low bits of SP in bits 31:30, a 1 in bit 29, a 0 in bit 28, the low 12 bits of the procedure entry mask in bits 27:16, and the PSW in bits 15:0 with T cleared is pushed on the stack. A longword 0 is pushed on the stack. FP is replaced by SP. AP is set to the saved SP (the value of the Stack Pointer after the number of arguments operand was pushed on the stack). The trap enables in the PSW are set to a known state. Integer overflow and decimal overflow are affected according to bits 14 and 15 of the entry mask, respectively; floating underflow is cleared; the T-bit is unaffected. AP is replaced by the saved SP. PC is replaced by the sum of destination operand and 2, which transfers control to the called procedure at the byte beyond the entry mask. The appearance of the stack after CALLS is executed is:


STACK
FRAME
:(SP)
:(FP)
(0 TO 3 BYTES SPECIFIED BY SPA)
NUMARG
:(AP)
NUMARG LONGWORDS OF ARGUMENT LIST

Notes:

1. If bits 13:12 of the entry mask are not 0, a reserved operand fault occurs.
2. On a reserved operand fault, the condition codes are unpredictable.
3. Normal use is to push the arglist onto the stack in reverse order prior to the CALLS. On return, the arglist is removed from the stack automatically.
4. The procedure calling standard and the condition handling facility require the following register saving conventions. R0 and R1 are always available for function return values and are never saved in the entry mask. All registers 2 through 11 which are modified in the called procedure must be preserved in the entry mask.

RET

RETURN FROM PROCEDURE

Purpose: transfer control from a procedure back to calling program

Format: opcode

Operation:

```
{restore SP from FP};
{restore registers};
{drop stack alignment};
{If CALLS, remove arglist};
{restore PSW};
```

Condition Codes:

N ← tmp1<3>;
Z ← tmp1<2>;
V ← tmp1<1>;
C ← tmp1<0>

Exceptions: Reserved operand

Opcodes: 04 RET Return From Procedure

Description: SP is replaced by FP plus 4. A longword containing stack alignment bits in bits 31:30, a CALLS/CALLG flag in bit 29, the low 12 bits of the procedure entry mask in bits 27:16, and a saved PSW in bits 15:0 is popped from the stack and saved in a temporary. PC, FP, and AP are replaced by longwords popped from the stack. A register restore mask is formed from bits 27:16 of the temporary. Scanning from bit 0 to bit 11 of the restore mask, the contents of registers whose number is indicated by set bits in the mask are replaced by longwords popped from the stack. SP is incremented by bits 31:30 of the temporary. PSW is replaced by bits 15:0 of the temporary. If bit 29 in the temporary is 1 (indicating that the procedure was called by CALLS), a longword containing the number of arguments is popped from the stack. Four times the unsigned value of the low byte of this longword is added to SP and SP is replaced by the result.

Notes:

1. A reserved operand fault occurs if tmp1<15:8> NEQ 0.
2. On a reserved operand fault, the condition codes are unpredictable. The value of tmp1<28> is ignored.
3. The procedure calling standard and condition handling facility assume that procedures which return a function value or a status code do so in R0 or R0 and R1.

CHAPTER 14

CHARACTER STRING INSTRUCTIONS AND THE CYCLIC REDUNDANCY CHECK

CHARACTER STRING INSTRUCTIONS

A character string is specified by two operands:

1. An unsigned word operand which gives the length of the character string in bytes.
2. The address of the lowest addressed byte of the character string. This is specified by a byte operand of address access type.

Each of the character string instructions uses general registers (R0 and R1, R0 through R3, or R0 through R5) to contain a control block which maintains updated addresses and state information during the execution of the instruction. At completion, these registers are available to software to use as string specification operands for a subsequent instruction. During the execution of the instructions, pending interrupt conditions are tested and if any are found, the control block is updated, a first part done (FPD) bit is set in the PSL, and the instruction is interrupted. After the interruption, the instruction resumes transparently. The format of the control block is:

	LENGTH 1	: R0
ADDRESS 1		: R1
	LENGTH 2	: R2
ADDRESS 2		: R3
	LENGTH 3	: R4
ADDRESS 3		: R5

The fields LENGTH 1, LENGTH 2, and LENGTH 3 (if required) contain the number of bytes remaining to be processed in the first, second, and third string operands respectively. The fields ADDRESS 1, ADDRESS 2, and ADDRESS 3 (if required) contain the address of the next byte to be processed in the first, second, and third string operands respectively.

MOVC

MOVE CHARACTER

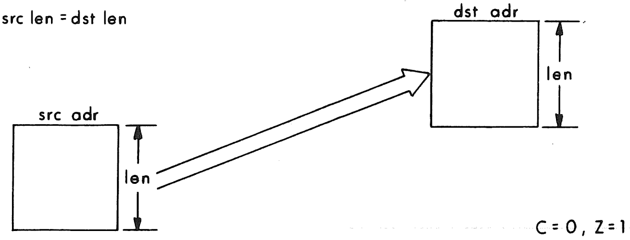
Purpose: to move character string or block of memory

Format:

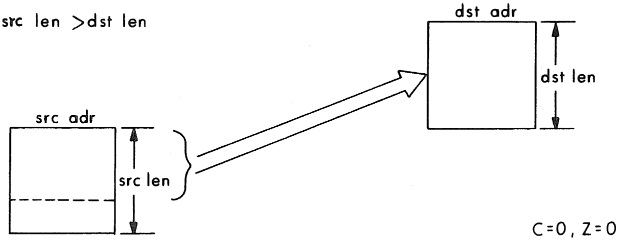
```
opcode len.rw, srcaddr.ab, dstaddr.ab          !3 operand
opcode srclen.rw, srcaddr.ab, fill.rb          !5 operand
dstlen.rw, dstaddr.ab
```

Operation:

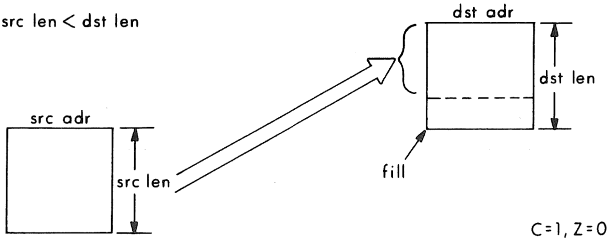
MOVC3,
MOVC5 If src len = dst len



MOVC5 If src len >dst len



MQVC5 If src len < dst len



Condition Codes:

N ← 0;
Z ← 1;
V ← 0;
C ← 0 !MOVC3

```
N ← srclen LSS dstlen;                       !MOVC5
Z ← srclen EQL dstlen;
V ← 0;
C ← srclen LSSU dstlen
```

Exceptions: None

Opcodes:

28	MOVC3	Move Character 3 Operand
2C	MOVC5	Move Character 5 Operand

Description: The destination string is replaced by the source string. If the destination string is longer than the source string, the highest address bytes of the destination are replaced by the fill operand; whereas, if the destination string is shorter that the source string, the highest addressed bytes of the source string are not moved. The operation of the instruction is such that overlap of the source and destination strings does not affect the result.

Notes:

1. After execution of MOVC3:

R0 =	0
R1 =	address of one byte beyond the source string
R2 =	0
R3 =	address of one byte beyond the destination string
R4 =	0
R5 =	0

2. After execution of MOVC5:

R0 =	number of unmoved bytes remaining in source string; R0 is nonzero only if source string is longer than destination string
R1 =	address of one byte beyond the last byte in source string that was moved
R2 =	0
R3 =	address of one byte beyond the destination string
R4 =	0
R5 =	0

3. MOVC3 is the preferred way to copy one block of memory to another.

4. MOVC5 with a 0 source length operand is the preferred way to fill a block of memory with the fill character.

MOVTC

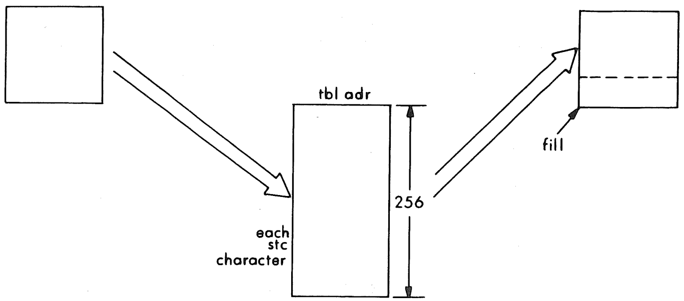
MOVE TRANSLATED CHARACTERS

Purpose: to move and translate character string

Format: opcode srclen.rw, srcaddr.ab, fill.rb, tbladdr.ab, dstlen.rw, dstaddr.ab

Operation:

MOVTC src len<dst len



NOTE: THE CASE OF src len = dst len AND src len>dst len SIMILAR TO THAT SHOWN IN THE MOVC5 INSTRUCTION

Condition Codes:
N ← srclen LSS dstlen;
Z ← srclen EQL dstlen;
V ← 0;
C ← srclen LSSU dstlen

Exceptions: None

Opcodes: 2E MOVTC Move Translated Characters

Description: The source string is translated and replaces the destination string. Translation is accomplished by using each byte of the source string as an index into a 256-byte table whose zeroth entry address is specified by the table address operand. The byte selected replaces the byte of the destination string. If the destination string is longer than the source string, the highest addressed bytes of the destination string are replaced by the fill operand. If the destination string is shorter than the source string, the highest addressed bytes of the source string are not translated and moved. The operation of the instruction is such that overlap of the source and destination strings does not affect the result. If the destination string overlaps the translation table, the destination string is unpredictable.

Notes: After execution:

R0 =	number of translated bytes remaining in source string; R0 is nonzero only if source string is longer than destination string
R1 =	address of one byte beyond the last byte in source string that was translated
R2 =	0
R3 =	address of the translation table
R4 =	0
R5 =	address of one byte beyond the destination string

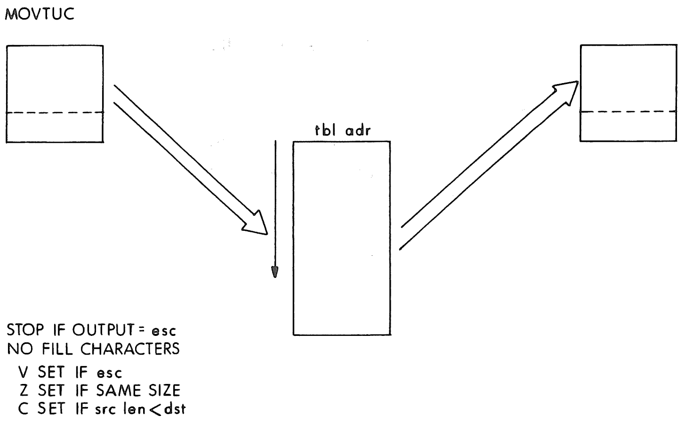
MOVTUC

MOVE TRANSLATED UNTIL CHARACTER

Purpose: To move and translate character string, handling escape codes

Format: opcode srclen.rw, srcaddr.ab, esc.rb, tbladdr.ab, dstlen.rw, dstaddr.ab

Operation:



Condition Codes:

N ← srclen LSS dstlen;
Z ← srclen EQL dstlen;
V ← {terminated by escape};
C ← srclen LSSU dstlen

Exceptions: None

Opcodes: 2F MOVTUC Move Translated Until Character

Description: The source string specified is translated and replaces the destination string. Translation is accomplished by using each byte of the source string as an index into a 256-byte table whose zeroth entry address is specified by the table address operand. The byte selected replaces the byte of the destination string. Translation continues until a translated byte is equal to the escape byte or until the source string or destination string is exhausted. If translation is terminated because of escape, the condition code V bit is set; otherwise, it is cleared. If the destination string overlaps the table, the destination string and R0 through R5 are unpredictable. If the source and destination strings overlap and their addresses are **not** identical, then, again, the destination string and R0 through R5 are unpredictable. If the source and destination string addresses are identical, the translation is performed correctly.

Notes: After execution:

R0 = number of bytes remaining in source string (including the byte which caused the escape); R0 is zero only if the entire source string was translated and moved without escape

R1 = address of the byte which resulted in destination string exhaustion or escape; or if no exhaustion or escape, R1 = address of one byte beyond the source string

R2 = 0

R3 = address of the table

R4 = number of bytes remaining in the destination string

R5 = address of the byte in the destination string which would have received the translated byte that caused the escape or would have received a translated byte if the source string were not exhausted; or if no exhaustion or escape, R1 = address of one byte beyond the destination string

CMPC

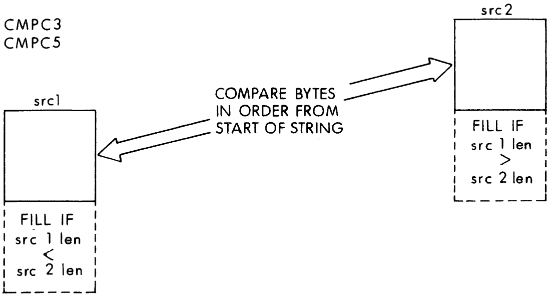
COMPARE CHARACTERS

Purpose: to compare two character strings

Format:

```
opcode len.rw, src1addr.ab, src2addr.ab          !3 operand
opcode src1len.rw, src1addr.ab, fill.rb          !5 operand
src2len.rw, src2addr.ab
```

Operation:


CMPC3
CMPC5
src1
src2
COMPARE BYTES IN ORDER FROM START OF STRING
FILL IF src 1 len < src 2 len
FILL IF src 1 len > src 2 len

NOTE: CONDITION CODES SET ON LAST COMPARE DONE

Condition Codes:

```
!Final condition codes reflect last affecting
!of condition codes in the operation.
N <- {first byte} LSS {second byte};
Z <- {first byte} EQL {second byte};
V <- 0;
C <- {first byte} LSSU {second byte}
```

Exceptions: None

Opcodes:

29	CMPC3	Compare Characters 3 Operand
2D	CMPC5	Compare Characters 5 Operand

Description: In 3-operand format, the bytes of string 1 specified by the length and address 1 operands are compared with the bytes of string 2 specified by the length and address 2 operands. Comparison proceeds until inequality is detected or until all the bytes of the strings have been examined. Condition codes are affected by the result of the last byte comparison. In 5-operand format, the bytes of the string 1 specified by the length 1 and address 1 operands are compared with the bytes of string 2 specified by the length 2 and address 2 operands. If one string is longer than the other, the shorter string is conceptually extended to the length of the longer by appending (at higher addresses) bytes equal to the fill operand. Comparison proceeds until inequality is detected or all the bytes of the strings have been examined. Condition codes are affected by the result of the last byte comparison.

Notes:

1. After execution of CMPC3:

   R0 = number of bytes remaining in string 1 (including byte which terminated comparison); R0 is zero only if strings are equal

   R1 = address of the byte in string 1 which terminated comparison; if strings are equal, R1 = address of one byte beyond string 1

   R2 = R0

   R3 = address of the byte in string 2 which terminated comparison: if strings are equal, R3 = address of one byte beyond string 2

2. After execution of CMPC5:

   R0 = number of bytes remaining in string 1 (including byte which terminated comparison); R0 is zero only if string 1 and string 2 are of equal length and equal or string 1 was exhausted before comparison terminated

   R1 = address of the byte in string 1 which terminated comparison; if comparison did not terminate before string 1 exhausted, R1 = address of one byte beyond string 1

   R2 = number of bytes remaining in string 2 (including byte which terminated comparison); R0 is zero only if string 2 and string 1 are of equal length or string 2 was exhausted before comparison terminated

R3 = address of the byte in string 2 which terminated comparison; if comparison did not terminate before string 2 was exhausted, R3 = address of one byte beyond string

3. If both strings have zero length, Z is set and N, V and C are cleared just as in the case of two equal strings.

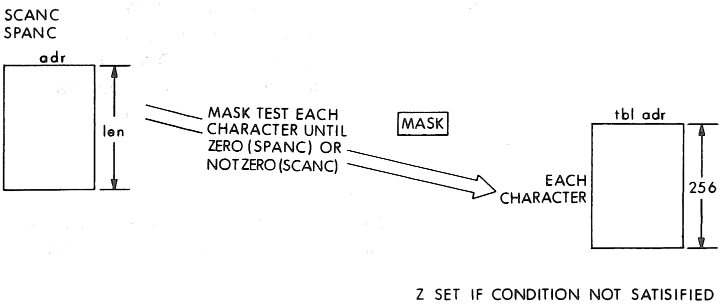
SCANC
SPANC

SCAN CHARACTERS, SPAN CHARACTERS

Purpose: to find or skip a set of characters in character string

Format: opcode len.rw, addr.ab, tbladdr.ab, mask.rb

Operation:


SCANC
SPANC
adr
len
MASK TEST EACH CHARACTER UNTIL ZERO (SPANC) OR NOTZERO (SCANC)
MASK
tbl adr
EACH CHARACTER
256
Z SET IF CONDITION NOT SATISIFIED

Condition Codes:
N ← 0;
Z ← R0 EQL 0;
V ← 0;
C ← 0

Exceptions: None

Opcodes:

2A	SCANC	Scan Characters
2B	SPANC	Span Characters

Description: The bytes of the string specified by the length and address operands are successively used to index into a 256-byte table whose zeroth entry address is specified by the table address operand. The byte selected from the table is ANDed with the mask operand. The operation continues until the result of the AND is nonzero for the SCANC instruction or zero for the SPANC instruction, or until all the bytes of the string have been exhausted. If a nonzero AND result for the SCANC or a zero result for the SPANC is detected, the condition code Z bit is cleared; otherwise, the Z bit is set.

Notes:

1. After execution:

   R0 = number of bytes remaining in the string (including the byte which produced the nonzero AND result for SCANC or zero result for SPANC);
   R0 is zero only if there was a zero AND result for SCANC or a nonzero result for SPANC

   R1 = address of the byte which produced nonzero AND result for SCANC or a zero AND result for SPANC; otherwise R1 = address of one byte beyond the string

   R2 = 0

   R3 = address of the table

2. If the string has zero length, condition code Z is set just as though the entire string were scanned (spanned).

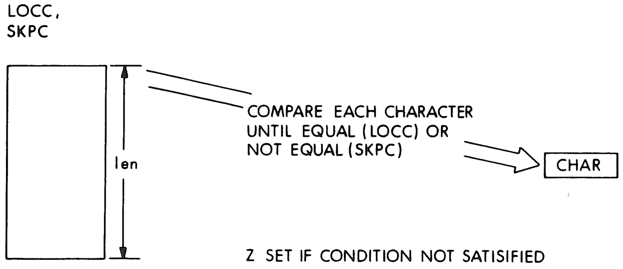
LOCC
SKPC

LOCATE CHARACTER, SKIP CHARACTER

Purpose: to find or skip character in character string

Format: opcode char.rb, len.rw, addr.ab

Operation:



Condition Codes:

N ← 0;
Z ← R0 EQL 0;
V ← 0;
C ← 0

Exceptions: None

Opcodes:

3A	LOCC	Locate Character
3B	SKPC	Skip Character

Description: The character operand is compared with the bytes of the string specified by the length and address operands. Comparison continues until equality is detected for the Locate Character instruction or inequality for the Skip Character instruction, or until all bytes of the string have been compared. If equality is detected for the Locate Character instruction, the condition code Z bit is cleared; otherwise the Z bit is set. If inequality is detected for the Skip Character instruction, the condition code Z bit is cleared; otherwise the Z bit is set.

Notes:

1. After execution:

R0 =	number of bytes remaining in the string (including located or unequal one) if (unequal) byte is located; otherwise R0 = 0
R1 =	address of the byte located if byte is located; otherwise R1 = address of one byte beyond the string

2. If the string has zero length, condition code Z is set just as though each byte of the entire string were equal (unequal) to the character.

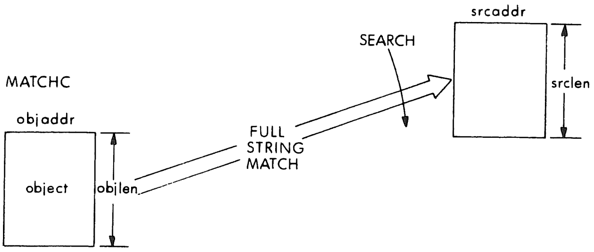
MATCHC

MATCH CHARACTERS

Purpose: to find substring (object) in character string

Format: opcode objlen.rw, objaddr.ab, srclen.rw, srcaddr.ab

Operation:



Condition Codes:

N ← 0;
Z ← R0 EQL 0; !match found
V ← 0;
C ← 0

Exceptions: None

Opcodes: 39 MATCHC Match Characters

Description: The source string specified by the source length and source address operands is searched for a substring which matches the object string specified by the object length and object address operands. If the substring is found, the condition code Z bit is set; otherwise, it is cleared.

Notes:

1. After execution:

R0 =	(if a match occurred) 0; otherwise R0 = the number of bytes in the object string
R1 =	(if a match occurred) the address of one byte beyond the object string; otherwise R1 = the address of the object string
R2 =	(if a match occurred) the number of bytes remaining in the source string after the match; otherwise R2 = 0.

R3 = (if a match occurred) the address of one byte beyond the last byte matched; otherwise R3 = the address of one byte beyond the source string.

2. If both strings have zero length or if the object string has zero length, condition code Z is set and R0 through R3 are left just as though the substring were found.
3. If the source string has zero length and the object string has nonzero length, condition code Z is cleared and R0 through R3 are left just as though the substring were not found.

CYCLIC REDUNDANCY CHECK INSTRUCTION

Cyclic Redundancy Checking (CRC) is an error detection method involving a division of the data stream by a CRC polynomial. The data stream is represented as a standard VAX string in memory. Error detection is accomplished by computing the CRC polynomial at the source and again at the destination, comparing the CRC computed at each end. The selected CRC polynomial should be such as to minimize the number of undetected block errors of specific lengths. Its choice is not given here, but can be found in references devoted to the topic.

The operands of the CRC instruction are a string descriptor, a 16-longword table, and an initial CRC. The string descriptor is a standard VAX operand pair of the length of the string in bytes (up to 65,535) and the starting address of the string. The contents of the table are a function of the CRC polynomial to be used. It can be calculated from the polynomial by a variety of algorithms. Several common CRC polynomials are also included below in Note 3. The initial CRC is used to start the polynomial correctly; typically, it has the value 0 or −1, but would be different if the data stream were represented by a sequence of noncontiguous strings.

The CRC instruction operates by scanning the string, and for each byte of the data stream, including it in the CRC being calculated. The byte is included by XORing it to the right eight bits of the CRC. Then the CRC is shifted right one bit, inserting zero on the left. The rightmost bit of the CRC (lost by the shift) is used to control the XORing of the CRC polynomial with the resultant CRC. If the bit is set, the polynomial is XORed with the CRC. Then the CRC is again shifted right and the polynomial is conditionally XORed with the result a total of eight times. Actual algorithms used can shift by one, two, or four bits at a time using the appropriate entries in a specially constructed table. The instruction produces a 32-bit CRC. For shorter polynomials, the result must be extracted from the 32-bit field. Data streams must be multiples of eight bits in length. If they are not, they must be right-adjusted in the string with leading 0 bits.

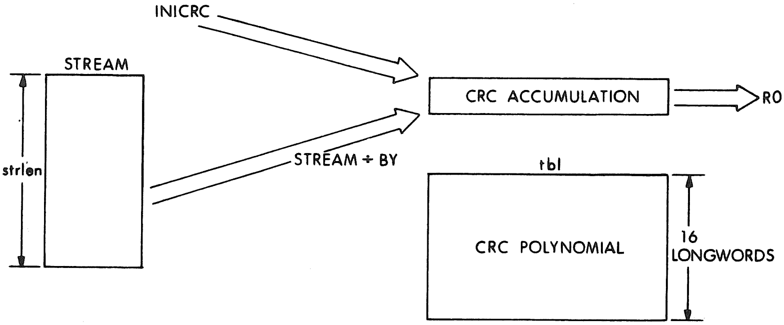
CRC

CALCULATE CYCLIC REDUNDANCY CHECK

Purpose: communications or software redundancy checks

Format: opcode tbl.ab, inicrc.rl, strlen.rw, stream.ab,

Operation:



Condition Codes:

N ← R0 LSS 0;
Z ← R0 EQL 0;
V ← 0;
C ← C

Exceptions: None

Opcodes: 0B CRC Calculate Cyclic Redundancy Check

Description: The CRC of the data stream described by the string descriptor is calculated. The initial CRC is given by inicrc and is normally 0 or −1 unless the CRC is calculated in several steps. R0 is replaced by the result. If the polynomial is less than order-32, the result must be extracted from R0. The CRC polynomial is expressed by the contents of the 16-longword table. See the notes for calculation of the table.

Notes:

1. If the data stream is not a multiple of eight bits long, it must be right-adjusted with leading zero fill.
2. If the CRC polynomial is less than order-32, the result must be extracted from the low-order bits of R0.
3. The following algorithm can be used to calculate the CRC table given a polynomial expressed as follows:

   poly<n> ←{coefficient of x**{order−1−n}}

   This routine is available as system library routine LIB$CRC_TABLE (poly.rl, table.ab). The table is the location of a 64-byte (16-longword) table into which the result will be written.

```
        SUBROUTINE LIB$CRC_TABLE (POLY,
        TABLE)
        INTEGER*4 POLY, TABLE(0:15), TMP, X
        DO 190 INDEX = 0, 15
        TMP = INDEX
        DO 150 I = 1,4
        X = TMP .AND. 1
        TMP = 1SHFT (TMP, -1)               !logical shift
                                            right one bit
        IF (X .EQ.1) TMP = TMP .XOR. POLY
            .
            .
            .
150     CONTINUE
        TABLE(INDEX) = TMP
190     CONTINUE
        RETURN
        END
```

4. The following are descriptions of some commonly used CRC polynomials.

CRC-16 (used in DDCMP and Bisync)

polynomial:	$x^{16} + x^{15} + x^2 + 1$
poly:	120001 (octal)
initialize:	0
result:	R0 <15:0>

CCITT (used in ADCCP, HDLC, SDLC)

polynomial:	$x^{16} + x^{12} + x^5 + 1$
poly:	102010 (octal)
initialize:	−1<15:0>
result:	complement of R0<15:0>

AUTODIN-II

polynomial	$x^{32} + x^{26} + x^{23} +$ $x^{22} + x^{16} +$ $x^{12} + x^{11} + x^{10} +$ $x^8 + x^7 + x^5 +$ $x^4 + x^2 + x + 1$
poly:	EDB88320 (hex)
initialize:	−1<31:0>
result:	complement of R0<31:0>

5. This instruction produces an unpredictable result unless the table is well formed, such as produced in Note 3. Note that for any well-formed table, entry [0] is always 0 and entry [8] is always the polynomial expressed as in Note 3. The operation can be implemented using shifts of one, two, or four bits at a time as follows:

	steps	index	table	
	per	table	multi-	use table
shift	byte	index	plier	entries
1	8	tmp3<0>	8	[0] = 0, <8>
2	4	tmp3<1:0>	4	[0] = 0, [4], [8], [12]
4	2	tmp3<3:0>	1	all

6. If the stream has zero length, the destination receives the initial CRC.

CHAPTER 15

DECIMAL STRING INSTRUCTIONS

Decimal string instructions operate on packed decimal strings. They treat decimal strings as integers, with the decimal point assumed immediately beyond the least significant digit of the string. If a string in which a result is to be stored is longer than the result, its most significant digits are filled with zeros.

Instructions are provided to convert between Packed Decimal and Trailing Numeric string (Overpunched or Zoned) and Leading Separate Numeric string formats. Where necessary, a specific data type is identified. Where the phase "decimal string" is used, it means any of these three data types.

A decimal string is specified by two operands:

1. For decimal strings, the length is the number of digits in the string. The number of bytes in the string is a function of the length and the type of decimal string referenced.
2. The address of the lowest addressed byte of the string. This byte contains the most significant digit for Trailing Numeric and Packed Decimal strings. This byte contains a sign for Leading Separate Numeric strings. The address is specified by a byte operand of address access type.

For more details on the Decimal String data type, see Chapter 4 of this Handbook.

Each of the decimal string instructions uses general registers 0 through 3 or 0 through 5 to contain a control block which maintains updated addresses and state during the execution of the instruction. At completion, the registers containing addresses are available to the software to use as string specification operands for a subsequent instruction on the same decimal strings.

During the execution of the instructions, pending interrupt conditions are tested, and if any are found the control block is updated. First Part Done (FPD) is set in the PSL, and the instruction is interrupted. After the interruption, the instruction resumes transparently. The format of the control block at completion is:

31 0	
0	:R0
ADDRESS 1	:R1
0	:R2
ADDRESS 2	:R3
0	:R4
ADDRESS 3	:R5

The fields ADDRESS 1, ADDRESS 2 and ADDRESS 3 (if required) contain the address of the byte containing the lowest addressed byte in the first, second, and third (if required) string operands, respectively.

Decimal Overflow

Decimal overflow, on the other hand, occurs if the destination string is too short to contain all the nonzero digits of the result. On overflow, the destination string is replaced by the correctly signed least significant digits of the result (even if the result is −0). Note that neither the high nibble of an even length Packed Decimal string, nor the sign byte of a Leading Separate Numeric string is used to store result digits.

Zero Numbers

A zero result has a positive sign for all operations that complete without decimal overflow. However, when digits are lost because of overflow, a zero result receives the sign (positive or negative) of the correct result.

A decimal string with value −0 is treated as identical to a decimal string with value +0. Thus, for example, +0 is equal to −0 in a Compare instruction. Similarly, when condition codes are affected on a −0 result they are affected as if the result were +0: i.e., N is cleared and Z is set.

Reserved Operand Exception

A reserved operand fault occurs if the length of a decimal string operand is outside the range 0 through 31, or if an invalid sign or digit is encountered in CVTSP and CVTTP.

Unpredictable Results

The result of any operation is unpredictable if any source decimal string operand contains invalid data. Except for CVTSP and CVTTP, the decimal string instructions do not verify the validity of source operand data.

If the destination operands overlap any source operands, the result of an operation will, in general, be unpredictable. Destination strings, registers used by the instruction, and condition codes will, in general, be unpredictable when a reserved operand fault occurs.

Packed Decimal Operations

Packed Decimal strings generated by the decimal string instructions always have the *preferred* sign representation: **12** for + and **13** for −, even though there are other options. An even length Packed Decimal string is always generated with a **0** digit in the high nibble of the first byte of the string.

A packed decimal string contains an invalid nibble if:

a) A digit occurs in the sign position;

b) A sign occurs in a digit position;

c) For an even length string, a nonzero nibble occurs in the high order nibble of the lowest addressed byte.

Zero Length Decimal Strings

The length of a Packed Decimal string can be zero. In this case, the value is zero (plus or minus) and one byte of storage is occupied. This byte must contain a **0** digit in the high nibble and the sign in the low nibble.

The length of a Trailing Numeric string can be zero. In this case, no storage is occupied by the string. If a destination operand is a zero length Trailing Numeric string, the sign of the operation is lost. Memory access faults will not occur when a zero length Trailing Numeric operand is specified, because no memory reference occurs.

The length of a Leading Separate Numeric string can be zero. In this case, one byte of storage is occupied by the sign. Memory is accessed when a zero length operand is specified, and a reserved operand fault will occur if an invalid sign is detected. The value of a zero length decimal string is identically zero.

MOVP

MOVE PACKED

Purpose: move a packed decimal string from one memory location to another

Format: opcode len.rw, srcaddr.ab, dstaddr.ab

Operation: ({dstaddr + ZEXT(len/2)}: dstaddr) ← ({srcaddr + ZEXT(len/2)}:srcaddr);

Condition Codes:

N ← {dst string} LSS 0;
Z ← {dst string} EQL 0;
V ← 0;
C ← C

Exceptions: Reserved operand

Opcodes:

34	MOVP	Move Packed

Description: The destination string specified by the length and destination address operands is replaced by the source string specified by the length and source address operands.

Notes:

1. After execution:

R0 =	0
R1 =	address of the byte containing the most significant digit of the source string
R2 =	0
R3 =	address of the byte containing the most significant digit of the destination string

2. The destinaton string, R0 through R3, and the condition codes are unpredictable if the destination string overlaps the source string, the source string contains an invalid nibble, or a reserved operand fault occurs.
3. If the source is −0, the result is +0, N is cleared and Z is set.

CMPP

COMPARE PACKED

Purpose: compare two packed decimal strings and set condition codes

Format:

```
opcode len.rw, scr1addr.ab, src2addr.ab          !3 operand
opcode src1len.rw, src1addr.ab,
src2len.rw, src2addr.ab                          !4 operand
```

Operation:

```
({src1addr + ZEXT(len/2)}:src1addr)=({src2addr +
ZEXT(len/2)}:src2addr);  !3 operand
({src1addr+ZEXT(src1len/2)}:src1addr)=({src2addr +
ZEXT(src2len/2)}:src2addr):  !4 operand
```

Condition Codes:

N ← {src1 string} LSS {src2 string};
Z ← {src1 string} EQL {src2 string};
V ← 0;
C ← 0

Exceptions: Reserved operand

Opcodes:

35	CMPP3	Compare Packed 3 Operand
37	CMPP4	Compare Packed 4 Operand

Description: In 3-operand format, the source 1 string specified by the length and source 1 address operands is compared to the source 2 string specified by the length and source 2 address operands. The only action is to affect the condition codes.

In 4-operand format, the source 1 string specified by the source 1 length and source 1 address operands is compared to the source 2 string specified by the source 2 length and source 2 address operands. The only action is to affect the condition codes.

Notes:

1. After execution of CMPP3 or CMPP4:

R0 =	0
R1 =	address of the byte containing the most significant digit of string 1
R2 =	0
R3 =	address of the byte containing the most significant digit of string 2

2. R0 through R3 and the condition codes are unpredictable if the source strings overlap, if either string contains an invalid nibble, or if a reserved operand fault occurs.

ADDP

ADD PACKED

Purpose: add one packed decimal string to another

Format:

```
opcode addlen.rw, addaddr.ab, sumlen.rw,
sumaddr.ab                                    !4 operand
opcode add1len.rw, add1addr.ab, add2len.rw.   !6 operand
add2addr.ab, sumlen.rw, sumaddr.ab
```

Operation:

```
{sum string} ← {sum string}                   !4 operand
+ {add string};
{sum string} ← {add1 string}                  !6 operand
+ {add2 string};
```

Condition Codes:

```
N ← {sum string} LESS 0;
Z ← {sum string} EQL 0;
V ← {decimal overflow};
C ← 0
```

Exceptions: Reserved operand
Decimal overflow

Opcodes:

20	ADDP4	Add Packed 4 Operand
21	ADDP6	Add Packed 6 Operand

Description: In 4-operand format, the addend string specified by the addend length and addend address operands is added to the sum string specified by the sum length and sum address operands and the sum string is replaced by the result.

In 6-operand format, the addend 1 string specified by the addend 1 length and addend 1 address operands is added to the addend 2 string specified by the addend 2 length and addend 2 address operands. The sum string specified by the sum length and sum address operands is replaced by the result.

Notes:

1. After execution of ADDP4:

   R0 = 0

   R1 = address of the byte containing the most significant digit of the addend string

   R2 = 0

   R3 = address of the byte containing the most significant digit of the sum string

2. After execution of ADDP6:

   R0 = 0

R1 = address of the byte containing the most significant digit of the addend1 string

R2 = 0

R3 = address of the byte containing the most significant digit of the addend2 string

R4 = 0

R5 = address of the byte containing the most significant digit of the sum string

3. The sum string, R0 through R3 (or R0 through R5 for ADD6), and the condition codes are unpredictable if the sum string overlaps the addend, addend1, or addend2 strings; the addend, addend1, addend2 or sum (4-operand only) strings contain an invalid nibble; or a reserved operand abort occurs.

SUBP

SUBTRACT PACKED

Purpose: subtract one packed decimal string from another

Format:

opcode sublen.rw, subaddr.ab, diflen.rw, difaddr.ab — !4 operand

opcode sublen.rw, subaddr.ab, minlen.rw, minaddr.ab, diflen.rw, difaddr.ab — !6 operand

Operation:

{dif string} ← {dif string} – {sub string}; — !4 operand
{dif string} ← {min string} – {sub string}; — !6 operand

Condition Codes:

N ← {dif string} LSS 0;
Z ← {dif string} EQL 0;
V ← {decimal overflow};
C ← 0

Exceptions:

Reserved operand
Decimal overflow

Opcodes:

22	SUBP4	Subtract Packed 4 Operand
23	SUBP6	Subtract Packed 6 Operand

Description: In 4-operand format, the subtrahend string, specified by subtrahend length and subtrahend address operands, is subtracted from the difference string, specified by the difference length and difference address operands, and the difference string is replaced by the result.

In 6-operand format, the subtrahend string, specified by the subtrahend length and subtrahend address operands, is subtracted from the minuend string, specified by the minuend length and minuend address operands. The difference string, specified by the difference length and difference address operands, is replaced by the result.

Notes:

1. After execution of SUBP4:

   R0 = 0

   R1 = address of the byte containing the most significant digit of the subtrahend string

   R2 = 0

   R3 = address of the byte containing the most significant digit of the difference string

2. After execution of SUBP6:

   R0 = 0

   R1 = address of the byte containing the most significant digit of the subtrahend string

   R2 = 0

   R3 = address of the byte containing the most significant digit of the minuend string

   R4 = 0

   R5 = address of the byte containing the most significant digit of the difference string

3. The difference string, R0 through R3 (R0 through R5 for SUBP6), and the condition codes are unpredictable if the difference string overlaps the subtrahend or minuend strings; the subtrahend, minuend, or difference (4-operand only) strings contain an invalid nibble; or a reserved operand abort occurs.

MULP

MULTIPLY PACKED

Purpose: multiply one packed decimal string by a second, result placed in a third

Format: opcode mulrlen.rw, mulraddr.ab, muldlen.rw, muldaddr.ab, prodlen.rw, prodaddr.ab

Operation: {prod string} ← {muld string} * {mulr string};

Condition Codes:
N ← {prod string} LSS 0;
Z ← {prod string} EQL 0;
V ← {decimal overflow};
C ← 0

Exceptions:
Reserved operand
Decimal overflow

Opcodes:

25	MULP	Multiply Packed

Description: The multiplicand string specified by the multiplicand length and multiplicand address operands is multiplied by the multiplier string specified by the multiplier length and multiplier address operands. The product string specified by the product length and product address operands is replaced by the result.

Notes:

1. After execution:

   R0 = 0

   R1 = address of the byte containing the most significant digit of the multiplier string

   R2 = 0

   R3 = address of the byte containing the most significant digit of the multiplicand string

   R4 = 0

   R5 = address of the byte containing the most significant digit of the product string

2. The product string, R0 through R5, and the condition codes are unpredictable if the product string overlaps the multiplier or multiplicand strings, the multiplier or multiplicand strings contain an invalid nibble, or a reserved operand abort occurs.

DIVP

DIVIDE PACKED

Purpose: divide one packed decimal string by a second, result placed in a third

Format: opcode divrlen.rw, divraddr.ab, divdlen.rw, divdaddr.ab, quolen.rw, quoaddr.ab

Operation: {quo string} ← {divd string} / {divr string};

Condition Codes:
N ← {quo string} LSS 0;
Z ← {quo string} EQL 0;
V ← {decimal overflow};
C ← 0

Exceptions:
Reserved operand
Decimal overflow
Divide by zero

Opcodes:

27	DIVP	Divide Packed

Description: The dividend string specified by the dividend length and dividend address operands is divided by the divisor string specified by the divisor length and divisor address operands. The quotient string specified by the quotient length and quotient address operands is replaced by the result.

Notes:

1. This instruction may allocate a 16-byte workspace on the stack. After execution, SP is restored to its original contents and the contents of {(SP) − 16} * {(SP) − 1} are unpredictable.
2. The division is performed such that:
   - The absolute value of the remainder (which is lost) is less than the absolute value of the divisor.
   - The product of the absolute value of the quotient and the absolute value of the divisor is less than or equal to the absolute value of the dividend.
   - The sign of the quotient is determined by the rules of algebra from the signs of the dividend and the divisor. If the value of the quotient is zero, the sign is always positive.
3. After execution:

   R0 = 0

   R1 = address of the byte containing the most significant digit of the divisor string

   R2 = 0

R3 = address of the byte containing the most significant digit of the dividend string

R4 = 0

R5 = address of the byte containing the most significant digit of the quotient string

4. The quotient string, R0 through R5, and the condition codes are unpredictable if the quotient string overlaps the divisor or dividend strings, the divisor or dividend string contains an invalid nibble, the divisor is 0, or a reserved operand abort occurs.

CVTLP

CONVERT LONG TO PACKED

Purpose: convert longword integer to packed decimal string

Format: opcode src.rl, dstlen.rw, dstaddr.ab

Operation: {dst string} ← conversion of src;

Condition Codes:
N ← {dst string} LSS 0;
Z ← {dst string} EQL 0;
V ← {decimal overflow};
C ← 0

Exceptions:
Reserved operand
Decimal overflow

Opcodes:

F9	CVTLP	Convert Long to Packed

Description: The source operand is converted to a packed decimal string and the destination string operand specified by the destination length and destination address operands is replaced by the result.

Notes:

1. After execution:

   R0 = 0

   R1 = 0

   R2 = 0

   R3 = address of the byte containing the most significant digit of the destination string

2. The destination string, R0 through R3, and the condition codes are unpredictable on a reserved operand abort.
3. Overlapping operands produce correct results.

Example:

```
MOVL #1234, R6
CVTLP R6, #3, DATA
```

The longword number 1234 is moved to register 6. Then 1234 is converted to a packed decimal string and replaces the 3-byte string whose most significant byte is represented by DATA.

CVTPL

CONVERT PACKED TO LONG

Purpose:

Format: opcode srclen.rw, srcaddr.ab, dst.wl

Operation: dst ← conversion of {src string}

Condition Codes:
N ← dst LSS 0;
Z ← dst EQL 0;
V ← {integer overflow};
C ← 0

Exceptions:
Reserved operand
Integer overflow

Opcodes:

36	CVTPL	Convert Packed to Long

Description: The source string specified by the source length and source address operands is converted to a longword and the destination operand is replaced by the result.

Notes:

1. After execution:

R0 =	0
R1 =	address of the byte containing the most significant digit of the source string
R2 =	0
R3 =	0

2. The destination operand, R0 through R3, and the condition codes are unpredictable on a reserved operand fault or if the string contains an invalid nibble.
3. The destination operand is stored after the registers are updated as specified in 1 above. Thus R0 through R3 may be used as the destination operand.
4. Integer overflow occurs if the source string has a value outside the range −2,147,483,648 through 2,147,483,647, and the destination operand is replaced by the low-order 32 bits of the correctly signed infinite-precision conversion. Thus, on overflow, the sign of the destination may be different from the sign of the source.
5. Overlapping operands produce correct results.

CVTPT

CONVERT PACKED TO TRAILING NUMERIC

Purpose: convert packed decimal string to trailing numeric string

Format: opcode srclen.rw, srcaddr.ab, tbladdr.ab, dstlen.rw, dstaddr.ab

Operation: {dst string} ← conversion of {src string};

Condition Codes:
N ← {src string} LSS 0;
Z ← {src string} EQL 0;
V ← {decimal overflow};
C ← 0

Exceptions:
Reserved operand
Decimal overflow

Opcodes:

24	CVTPT	Convert Packed to Trailing Numeric

Description: The source packed decimal string specified by the source length and source address operands is converted to a trailing numeric string. The destination string specified by the destination length and destination address operands is replaced by the result. The condition code N and Z bits are affected by the value of the source packed decimal string.

Conversion is effected by using the highest addressed byte of the source string (i.e., the byte containing the sign and the least significant digit) as an unsigned index into a 256-byte table whose zeroth entry address is specified by the table address operand. The byte read out of the table replaces the least significant byte of the destination string. The remaining bytes of the destination string are replaced by the ASCII representations of the values of the corresponding packed decimal digits of the source string.

Notes:

1. After execution:

R0 =	0
R1 =	address of the byte containing the most significant digit of the source string
R2 =	0
R3 =	address of the most significant digit of the destination string

2. The destination string, R0 through R3, and the condition codes are unpredictable if the destination string overlaps the source string or the table, the source string or the table contains an invalid nibble, or a reserved operand abort occurs.

3. The condition codes are computed on the value of the source string even if overflow results. In particular, condition code N is set if and only if the source is nonzero and contains a minus sign.
4. By appropriate specification of the table, conversion to any form of trailing numeric string may be realized. See Chapter 4 for the preferred form of trailing overpunch, zoned, and unsigned data. In addition, the table may be set up for absolute value, negative absolute value, or negative conversions.
5. Decimal overflow occurs if the destination string is too short to contain the converted result of a nonzero packed decimal source string (not including leading zeroes). Conversion of a source string with zero value never results in overflow. Conversion of a nonzero source string to a zero length destination string results in overflow.
6. If decimal overflow occurs, the value stored in the destination may be different from the value indicated by the condition codes (Z and N bits).

CVTTP

CONVERT TRAILING NUMERIC TO PACKED

Purpose: convert trailing numeric string to packed decimal string

Format: opcode srclen.rw, srcaddr.ab, tbladdr.ab, dstlen.rw, dstaddr.ab

Operation: {dst string} ← conversion of {src string};

Condition Codes:

N ← {dst string} LSS 0;
Z ← {dst string} EQL 0;
V ← {decimal overflow};
C ← 0

Exceptions:

Reserved operand
Decimal overflow

Opcodes:

26	CVTTP	Convert Trailing Numeric to Packed

Description: The source Trailing Numeric string specified by the source length and source address operands is converted to a packed decimal string and the destination packed decimal string specified by the destination address and destination length operands is replaced by the result.

Conversion is effected by using the highest addressed (trailing) byte of the source string as an unsigned index into a 256-byte table whose zeroth entry is specified by the table address operand. The byte read out of the table replaces the highest addressed byte of the destination string (i.e., the byte containing the sign and the least significant digit). The remaining packed digits of the destination string are replaced by the low order four bits of the corresponding bytes in the source string.

Notes:

1. A reserved operand abort occurs if:
   - The length of the source Trailing Numeric string is outside the range 0 through 31.
   - The length of the destination packed decimal string is outside the range 0 through 31.
   - The source string contains an invalid byte. An invalid byte is any value other than ASCII "0" through "9" in any high-order byte (i.e., any byte except the least significant byte).
   - The translation of the least significant digit produces an invalid packed decimal digit or sign nibble.
2. After execution:

R0 =	0
R1 =	address of the most significant digit of the source string

R2 = 0

R3 = address of the byte containing the most significant digit of the destination string

3. The destination string, R0 through R3, and the condition codes are unpredictable if the destination string overlaps the source string or the table, or a reserved operand fault occurs.
4. If the Convert instruction produces a −0 without overflow, the destination packed decimal string is changed to a +0 representation, condition code N is cleared, and Z is set.
5. If the length of the source string is 0, the destination packed decimal string is set identically equal to 0, and the translation table is not referenced.
6. By appropriate specification of the table, conversion from any form of Trailing Numeric string may be realized. See Chapter 4 for the preferred form of trailing overpunch, zoned, and unsigned data. In addition, the table may be set up for absolute value, negative absolute value or negated conversions.
7. If the table translation produces a sign nibble containing any valid sign, the preferred sign representation is stored in the destination packed decimal string.

CVTPS

CONVERT PACKED TO LEADING SEPARATE NUMERIC

Purpose: convert packed decimal string to leading separate numeric string

Format: opcode srclen.rw, srcaddr.ab, dstlen.rw, dstaddr.ab

Operation: {dst string} ← conversion of {src string};

Condition Codes:
N ← {src string} LSS 0;
Z ← {src string} EQL 0;
V ← {decimal overflow};
C ← 0

Exceptions:
Reserved operand
Decimal overflow

Opcodes:

08	CVTPS	Convert Packed to Leading Separate Numeric

Description: The source packed decimal string specified by the source length and source address operands is converted to a Leading Separate Numeric string. The destination string specified by the destination length and destination address operands is replaced by the result.

Conversion is effected by replacing the lowest addressed byte of the destination string by the ASCII character + or −, determined by the sign of the source string. The remaining bytes of the destination string are replaced by the ASCII representations of the values of the corresponding packed decimal digits of the source string.

Notes:

1. After execution:

R0 =	0
R1 =	address of the byte containing the most significant digit of the source string
R2 =	0
R3 =	address of the sign byte of the destination string

2. The destination string, R0 through R3, and the condition codes are unpredictable if the destination string overlaps the source string, the source string contains an invalid nibble, or a reserved operand abort occurs.
3. This instruction produces an ASCII "+" or "−" in the sign byte of the destination string.
4. If decimal overflow occurs, the value stored in the destination may be different from the value indicated by the condition codes (Z and N bits).

5. If the conversion produces a −0 without overflow, the destination Leading Separate Numeric string is changed to a +0 representation.

CVTSP

CONVERT LEADING SEPARATE NUMERIC TO PACKED

Purpose: convert leading separate numeric string to packed decimal string

Format: opcode srclen.rw, srcaddr.ab, dstlen.rw, dstaddr.ab

Operation: {dst string} ← conversion of {src string};

Condition Codes:

N ← {dst string} LSS 0;
Z ← {dst string} EQL 0;
V ← {decimal overflow};
C ← 0

Exceptions:

Reserved operand
Decimal overflow

Opcodes: 09 CVTSP Convert Leading Separate Numeric to Packed

Description: The source numeric string specified by the source length and source address operands is converted to a packed decimal string, and the destination string specified by the destination address and destination length operands is replaced by the result.

Notes:

1. A reserved operand fault occurs if;
   - The length of the source Leading Separate Numeric string is outside the range 0 through 31.
   - The length of the destination packed decimal string is outside the range 0 through 31.
   - The source string contains an invalid byte. An invalid byte is any character other than an ASCII "0" through "9" in a digit byte or an ASCII "+," "<space>," or "−" in the sign byte.
2. After execution:

R0 =	0
R1 =	address of the sign byte of the source string
R2 =	0
R3 =	address of the byte containing the most significant digit of the destination string

3. The destination string, R0 through R3, and the condition codes are unpredictable if the destination string overlaps the source string, or a reserved operand abort occurs.

ASHP

ARITHMETIC SHIFT AND ROUND PACKED

Purpose: scale numeric content of a packed decimal string by a power of 10

Format: opcode cnt.rb, srclen.rw, srcaddr.ab, round.rb, dstlen.rw, dstaddr.ab

Operation: {dst string} ← {{src string} + {round <3:0> * (10 ** (−cnt−1))}} * (10 ** cnt);

Condition Codes:
N ← {dst string} LSS 0;
Z ← {dst string} EQL 0:
V ← {decimal overflow};
C ← 0

Exceptions:
Reserved operand
Decimal overflow

Opcodes: F8 ASHP Arithmetic Shift and Round Packed

Description: The source string specified by the source length and source address operands is scaled by a power of 10 specified by the count operand. The destination string specified by the destination length and destination address operands is replaced by the result.

A positive count operand effectively multiplies; a negative count effectively divides; and a zero count just moves and affects condition codes. When a negative count is specified, the result is rounded using the round operand.

Notes:

1. After execution:

   R0 = 0

   R1 = address of the byte containing the most significant digit of the source string

   R2 = 0

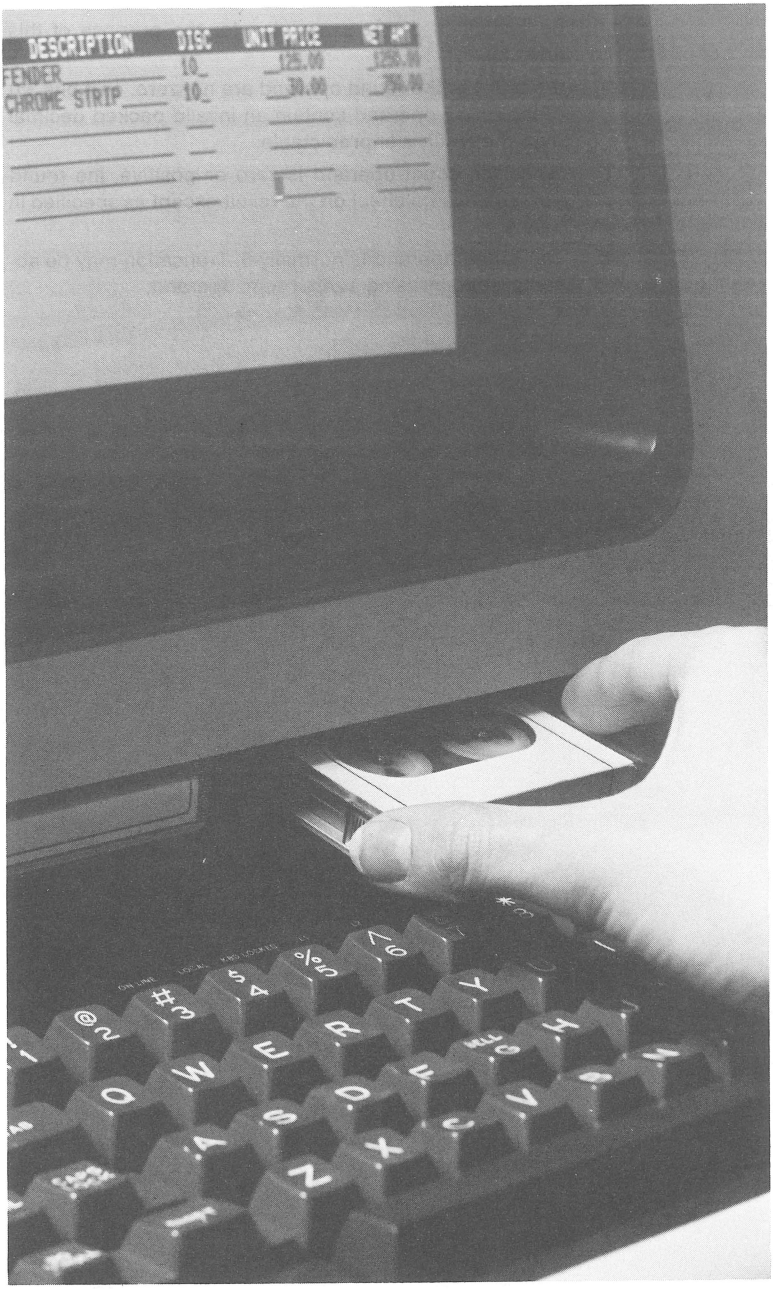
   R3 = address of the byte containing the most significant digit of the destination string

2. The destination string, R0 through R3, and the condition codes are unpredictable if the destination string overlaps the source string, the source string contains an invalid nibble, or a reserved operand abort occurs.

3. When the count operand is negative, the result is rounded by decimally adding bits 3:0 of the round operand to the most significant low order digit discarded and propagating the carry, if any, to higher order digits. Both the source operand and the round operand are considered to

be quantities of the same sign for the purpose of this addition.

4. If bits 7:4 of the round operand are nonzero, or if bits 3:0 of the round operand contain an invalid packed decimal digit, the result is unpredictable.
5. When the count operand is zero or positive, the round operand has no effect on the result except as specified in note 4.
6. The round operand is normally 5. Truncation may be accomplished by using a zero round operand.


DESCRIPTION
DISC
UNIT PRICE
NET AMT
FENDER
10_
125.00
CHROME STRIP
10_
30.00

CHAPTER 16
EDIT INSTRUCTION (EDITPC)

The edit instruction, naturally enough, implements the common editing functions which occur in handling fixed format output. It operates by converting a packed decimal string (input) to a character string (output), generating characters for the output. It is exemplified by a MOVE to a numeric edited (PICTURE) item in COBOL or PL/I, but the instruction can also be used for other applications. When converting digits, options include leading zero fill, leading zero protection, insertion of floating sign, of floating currency symbol, or of special sign representations, and blanking an entire field when it is zero.

The operands to the EDITPC instruction are an input packed decimal string descriptor, a pattern specification, and the starting address of the output string. The packed decimal descriptor comprises a standard VAX operand pair of the length of the decimal string in digits (up to 31) and the starting address of the string. The pattern specification is the starting address of a "pattern operation editing sequence" which is interpreted in much the same way as the normal instructions are. Only the starting address of the output string is required, because the pattern defines the length unambiguously.

While the EDITPC instruction is operating, it manipulates two character registers and the four condition codes. One character register contains the fill character. This is normally an ASCII blank, but would be changed to * for check protection. The other character register contains the sign character, initially either an ASCII blank or a – sign, depending upon the sign of the input. It can be changed to allow other sign representations, such as ± or plus/blank and can be manipulated in order to output special notations such as **CR** or **DB**. The sign register can also be changed to the currency sign in order to implement a floating currency sign. After execution, the condition codes note the sign of the input (N), the presence of a nonzero source (Z), an overflow condition (V), and the presence of significant digits (C). Condition code N is determined at the start of the instruction and is not changed thereafter (except for correcting a –0 input). The other condition codes are computed and updated as the instruction proceeds. When the EDITPC instruction terminates, registers 0 through 5 contain the conventional values after a decimal instruction.

Following the description of EDITPC, the edit instruction, we define the twelve edit pattern operators in a fashion similar to that used earlier in describing the members of the instruction set. For these purposes, the operand is either a repeat count (r) from 1 through 15, an unsigned byte length (len), or a character byte (ch).

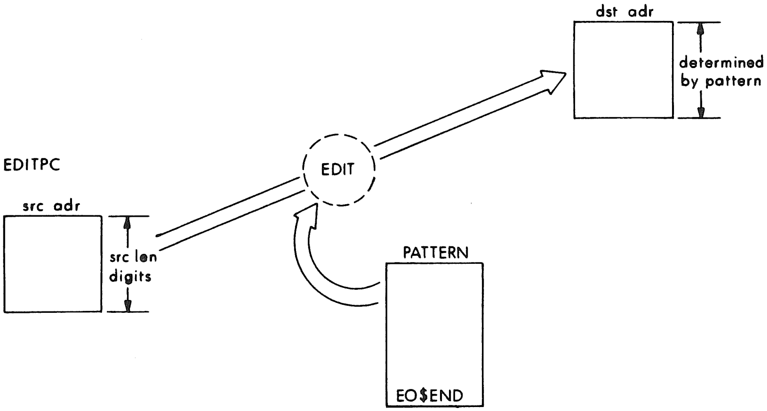
EDITPC

EDIT PACKED TO CHARACTER STRING

Purpose: edit source string

Format: opcode srclen.rw, srcaddr.ab, pattern.ab, dstaddr.ab

Operation:



Condition Codes:

N ← {src string} LSS 0; ! N ← 0 if src is −0
Z ← {src string} EQL 0;
V ← {decimal overflow}; ! nonzero digits lost
C ← {significance}

Exceptions: Reserved operand
Decimal overflow

Opcodes:

38	EDITPC	Edit Packed to Character String

Description: The destination string specified by the pattern and destination address operand is replaced by the edited version of the source string specified by the source length and source address operands. Editing is performed according to the pattern string, starting at the address pattern and extending until a pattern end (EO$END) pattern operator is encountered. The pattern string consists of one-byte pattern operators. Some pattern operators take no operands; some take a repeat count which is contained in the rightmost nibble of the pattern operator itself; the rest take a one-byte operand which follows the pattern operator immediately. This operand is either an unsigned integer length or a byte character. *The individual pattern operators are described on subsequent pages.*

Notes:

1. A reserved operand fault occurs with FPD cleared if srclen GTRU 31.
2. The destination string is unpredictable if the source string contains an invalid nibble, if the EO$ADJUST_INPUT operand is outside the range 1 through 31, if the source and destination strings overlap, or if the pattern and destination strings overlap.
3. After execution:

R0 =	length of source string
R1 =	address of the byte containing the most significant digit of the source string
R2 =	0
R3 =	address of the byte containing the EO$END pattern operator
R4 =	0
R5 =	address of one byte beyond the last byte of the destination string

   If the destination string is unpredictable, R0 through R5 and the condition codes are unpredictable.

4. If V is set at the end and DV is enabled, numeric overflow trap occurs unless the conditions in Note 9 are satisfied.
5. The destination length is specified exactly by the pattern operators in the pattern string. If the pattern is incorrectly formed or if it is modified during the execution of the instruction, the length of the destination string is unpredictable.
6. If the source is −0, the result may be −0 unless a fixup pattern operator is included (EO$BLANK_ZERO or EO$REPLACE_SIGN).
7. The contents of the destination string and the memory preceding it are unpredictable if the length covered by EO$BLANK_ZERO or EO$REPLACE_SIGN is 0 or is outside the destination string.
8. If more input digits are requested by the pattern than are specified, then a reserved operand abort is taken with R0 = −1 and R3 = location of pattern operator which requested the extra digit. The condition codes and other registers are as specified in Note 11. This abort is not continuable.

9. If fewer input digits are requested by the pattern than are specified, then a reserved operand abort is taken with R3 = location of EO$END pattern operator. The condition codes and other registers are as specified in Note 11. This abort is not continuable.

10. On an unimplemented or reserved pattern operator, a reserved operand fault is taken with R3 = location of the faulting pattern operator. The condition codes and other registers are as specified in Note 11. This fault is continuable as long as the defined register state is manipulated according to the pattern operator description and the other state specified is preserved.

11. On a reserved operand exception as specified in Notes 8 through 10, FPD is set and the condition codes and registers are as follows:

    N = {src has minus sign}

    Z = all source digits 0 so far

    V = nonzero digits lost

    C = significance

    R0 = −zeros <15:0>'srclen<15:0>

    R1 = current source location

    R2 = ???'sign' fill

    R3 = edit pattern operator causing exception

    R4 = ???

    R5 = location of next destination byte

    where:
    zeros = count of source zeros to supply
    sign = current contents of sign character register
    fill = current contents of fill character register

SUMMARY OF EDIT PATTERN OPERATORS

	Name	Operand	Summary
Insert:	EO$INSERT	ch	insert character, fill if insignificant
	EO$STORE_SIGN	—	insert sign
	EO$FILL	r	insert fill
Move:	EO$MOVE	r	move digits, filling insignificant
	EO$FLOAT	r	move digits, floating sign
	EO$END_FLOAT	—	end floating sign
Fixup:	EO$BLANK-ZERO	len	fill backward when zero
	EO$REPLACE_SIGN	len	replace with fill if −0
Load:	EO$LOAD_FILL	ch	load fill character
	EO$LOAD_SIGN	ch	load sign character
	EO$LOAD_PLUS	ch	load sign character if positive
	EO$LOAD_MINUS	ch	load sign character if negative
Control:	EO$SET_SIGNIF	—	set significance flag
	EO$CLEAR_SIGNIF	—	clear significance flag
	EO$ADJUST_INPUT	len	adjust source length
	EO$END	—	end edit

where: ch = one character
r = repeat counter in the range 1 through 15
len = length in the range 1 through 255

EDIT PATTERN OPERATOR ENCODING

(hex)	
00	EO$END
01	EO$END_FLOAT
02	EO$CLEAR_SIGNIF
03	EO$SET_SIGNIF
04	EO$STORE_SIGN
40	EO$LOAD_FILL
41	EO$LOAD_SIGN

42	EO$LOAD_PLUS	character is in next byte
43	EO$LOAD_MINUS	
44	EO$INSERT	
45	EO$BLANK_ZERO	
46	EO$REPLACE_SIGN	unsigned length is in next byte
47	EO$ADJUST_INPUT	
60...7F	Reserved to DIGITAL's CSS, customers	
81...8F	EO$FILL	
91...9F	EO$MOVE	repeat count is <3:0>
A1...AF	EO$FLOAT	

In the formal descriptions, the following two routines are invoked.

```
READ:           !function value 0 through 9

          if R0 EQL 0        then {reserved operand}
          if R0 LSS 0 then
                             begin
                             READ ← 0;
                             R0 <31:16> ←R0<31:16> + 1;
                             !see EO$ADJUST_INPUT
                             end;
          else
                             begin
                             READ ← (R1)<3+4*R0<0>:4*R0<0>>;
                             !get next nibble
                             !alternating high then low
                             R0 ← R0 − 1;
                             if R0<0> EQL 1 then R1 ← R1 + 1;
                             end;
          return;

STORE (char):   (R5) ← char;
                R5 ← R5 + 1;
                return;
```

Also the following definitions are used:

```
                    fill = R2<7:0>
                    sign = R2<15:8>
```

EO$INSERT

INSERT CHARACTER

Purpose: insert a fixed character, substituting the fill character if not significant

Format: pattern ch

Operations: if PSW<C>EQL 1 then STORE (ch) else STORE (fill);

Pattern Operators:

44	EO$INSERT	Insert Character

Description: The pattern operator is followed by a character. If sigificance is set, then the character is placed into the destination. If significance is not set, then the contents of the fill register are placed into the destination.

Notes: This pattern operator is used for blankable inserts (e.g., comma) and fixed inserts (e.g., slash). Fixed inserts require that significance be set (by EO$SET_SIGNIF or EO$END_FLOAT).

EO$STORE_SIGN

STORE SIGN

Purpose: insert the sign character

Format: pattern

Operations: STORE (sign);

Pattern Operators: 04 EO$STORE_SIGN Store Sign

Description: The contents of the sign register are placed into the destination.

Notes: This pattern operator is used for any nonfloating arithmetic sign. It should be preceded by a EO$LOAD_PLUS and/or EO$LOAD_MINUS if the default sign convention is not desired.

EO$FILL

STORE FILL

Purpose:	insert the fill character
Format:	pattern r
Operations:	repeat r do STORE (fill);
Pattern Operators:	8x EO$FILL Store Fill
Description:	The right nibble of the pattern operator is the repeat count. The contents of the fill register are placed into the destination repeat times.
Notes:	This pattern operator is used for fill (blank) insertion.

EO$MOVE

MOVE DIGITS

Purpose: move digits, filling for insignificant digits (leading zeros)

Format: pattern r

Operations:

```
repeat r do
     begin
     tmp ← READ;
     if tmp NEQU 0 then
       begin
       PSW<Z> ← 0;
       PSW<C> ← 1;    !set significance
       end;
     if PSW<C> EQL 0 then STORE (fill)
       else STORE ("0" + tmp);
     end;
```

Pattern Operators: 9x EO$MOVE Move Digits

Description: The right nibble of the pattern operator is the repeat count. For repeat times, the following algorithm is executed: the next digit is moved from the source to the destination; a) if the digit is nonzero, significance is set and zero is cleared; b) if the digit is not significant (i.e., is a leading zero), it is replaced by the contents of the fill register in the destination.

Notes:

1. If r is greater than the number of digits remaining in the source string, a reserved operand abort is taken.
2. This pattern operator is used to move digits without a floating sign. If leading zero suppression is desired, significance must be clear. If leading zero should be explicit, signifance must be set. A string of EO$MOVEs intermixed with EO$INSERTs and EO$FILLs will handle suppression correctly.
3. If check protection (*) is desired, EO$LOAD_FILL must precede the EO$MOVE.

EO$FLOAT

FLOAT SIGN

Purpose: move digits, floating the sign across insignificant digits

Format: pattern r

Operations:

```
repeat r do
    begin
    tmp ← READ;
    if tmp NEQU 0 then
     begin
     if PSW<C> EQL 0 then STORE (sign);
     begin
     STORE (sign);
     PSW<Z> ← 0;
     PSW<C> ← 1;   !set significance
     end;
      end;       if PSW<C> EQL 0 then STORE (fill)
     else STORE ("0" + tmp);
    end;
```

Pattern Operators: Ax EO$FLOAT Float Sign

Description: The right nibble of the pattern operator is the repeat count. For repeat times, the following algorithm is executed: the next digit from the source is examined; a) if it is nonzero and significance is not yet set, then the contents of the sign register are stored in the destination, significance is set, and zero is cleared; b) if the digit is significant, it is stored in the destination, otherwise the content of the fill register is stored in the destination.

Notes:

1. If r is greater than the number of digits remaining in the source string, a reserved operand abort is taken.
2. This pattern operator is used to move digits with a floating arithmetic sign. The sign must already be set up as for EO$STORE_SIGN. A sequence of one or more EO$FLOATs can include intermixed EO$INSERTs and EO$FILLs. Significance must be clear before the first pattern operator of the sequence. The sequence must be terminated by one EO$END_FLOAT.
3. This pattern operator is used to move digits with a floating currency sign. The sign must already be set up with an EO$LOAD_SIGN. A sequence of one or more EO$FLOATs can include intermixed EO$INSERTs and EO$FILLs. Significance must be clear before the first pattern operator of the sequence. The sequence must be terminated by one EO$END_FLOAT.

EO$END_FLOAT

END FLOATING SIGN

Purpose: end a floating sign operation

Format: pattern

Operations:

```
if PSW<C> EQL 0 then
        begin
        STORE (sign);
        PSW<C> ← 1;    !set significance
        end;
```

Pattern Operators: 01 EO$END_FLOAT End Floating Sign

Description: If the floating sign has not yet been placed in the destination (i.e., if significance is not set), the contents of the sign register are stored in the destination and significance is set.

Notes: This pattern operator is used after a sequence of one or more EO$FLOAT pattern operators which start with significance clear. The EO$FLOAT sequence can include intermixed EO$INSERTs and EO$FILLs.

EO$BLANK_ZERO

BLANK BACKWARDS WHEN ZERO

Purpose: fix up the destination to be blank when the value is zero

Format: pattern len

Operations:

```
if len EQLU 0 then {UNPREDICTABLE};
if PSW<Z> EQL 1 then
      begin
      R5 <- R5 - len;
      repeat len do STORE (fill);
      end;
```

Pattern Operators: 45 EO$BLANK_ZERO Blank Backwards When Zero

Description: The pattern operator is followed by an unsigned byte integer length. If the value of the source string is zero, then the contents of the fill register are stored into the last length bytes of the destination string.

Notes:

1. The length must be nonzero and within the destination string already produced. If it is not, the contents of the destination string and the memory preceding it are unpredictable.
2. This pattern operator is used to blank out any characters stored in the destination under a forced significance, such as a sign or the digits following the radix point.

EO$REPLACE_SIGN

REPLACE SIGN WHEN MINUS ZERO

Purpose: fix up the destination sign when the value is minus zero

Format: pattern len

Operations:

```
if len EQLU 0 then {UNPREDICTABLE};
if PSW<Z> EQL 1 and PSW<N> EQL 1 then
      (R5 – len) ← fill
```

Pattern Operators: 46 EO$REPLACE_SIGN Replace Sign When Minus Zero

Description: The pattern operator is followed by an unsigned byte integer length. If the value of the source string is zero (i.e., if Z is set), then the contents of the fill register are stored into the byte of the destination string which is "length" bytes before the current position.

Notes:

1. The length must be nonzero and within the destination string already produced. If it is not, the contents of the destination string and the memory preceding it are unpredictable.
2. This pattern operator is used to correct a stored sign (EO$END_FLOAT or EO$STORE_SIGN) if a minus was stored and the source value turned out to be zero.

EO$LOAD

LOAD REGISTER

Purpose: change the contents of the fill or sign register

Format: pattern ch

Operations:

```
                                  !select one depending on pattern operator
fill ← ch;                        !EO$LOAD_FILL
sign ← ch;                        !EOSLOAD_SIGN
if PSW<N> EQL 0 then sign ← ch;   !EO$LOAD_PLUS
if PSW<N> EQL 1 then sign ← ch;   !EO$LOAD_MINUS
```

Pattern Operators:

40	EO$LOAD_FILL	Load Fill Register
41	EO$LOAD_SIGN	Load Sign Register
42	EO$LOAD_PLUS	Load Sign Register If Plus
43	EO$LOAD_MINUS	Load Sign Register If Minus

Description: The pattern operator is followed by a character. For EO$LOAD_FILL, this character is placed into the fill register. For EO$LOAD_SIGN, this character is placed into the sign register if the source string has a positive sign. For EO$LOAD_MINUS, this character is placed into the sign register if the source string has a negative sign.

Notes:

1. EO$LOAD_FILL is used to set up check protection (* instead of space).
2. EO$LOAD_SIGN is used to set up a floating currency sign.
3. EO$LOAD_PLUS is used to set up a nonblank plus sign.
4. EO$LOAD_MINUS is used to set up a nonminus minus sign (such as CR, DB, or the PL/I+).

EO$_SIGNIF

SIGNIFICANCE

Purpose: control the significance (leading zero) indicator

Format: pattern

Operations:

```
PSW<C> ← 0;      !EO$CLEAR_SIGNIF
PSW<C> ← 1;      !EO$SET_SIGNIF
```

Pattern Operators:

02	EO$CLEAR_SIGNF	Clear Significance
03	EO$SET_SIGNIF	Set Significance

Description: The significance indicator is set or cleared. This controls the treatment of leading zeros (leading zeros are zero digits for which the significance indicator is clear).

Notes:

1. EO$CLEAR_SIGNIF is used to initialize leading zero suppression (EO$MOVE) or floating sign (EO$FLOAT) following a fixed insert (EO$INSERT with significance set).
2. EO$SET_SIGNIF is used to avoid leading suppression (before EO$MOVE) or to force a fixed insert (before EO$INSERT).

EO$ADJUST_INPUT

ADJUST INPUT LENGTH

Purpose: handle source strings with lengths different from the output

Format: pattern len

Operations:

```
if len EQLU 0 or len GTRU 31 then {UNPREDICTABLE};
if R0 <15:0> GTRU len
then
      begin
      R0<31:16> ← 0
      repeat R0<15:0> – len do
         if READ NEQU 0 then
         begin
         PSW<Z> ← 0;
         PSW<C>←1;             !set significance
         PSW<V> ← 1;
         end;
      end;
else R0<31:16> ← R0<15:0> – len;
!negative of number to fill
```

Pattern Operators: 47 EO$ADJUST_INPUT Adjust Input Length

Description: The pattern operator is followed by an unsigned byte integer length in the range 1 through 31. If the source string has more digits than this length, the excess digits are read and discarded. If any discarded digits are nonzero, then overflow is set, significance is set, and zero is cleared. If the source string has fewer digits than this length, a counter is set to the number of leading zeros to supply. This counter is stored as a negative numer in R0<31:16>.

Notes: If length is not in the range 1 through 31, the destination string, condition codes, and R0 through R5 are unpredictable.

EO$END

END EDIT

Purpose: end the edit operation

Format: pattern

Operations:

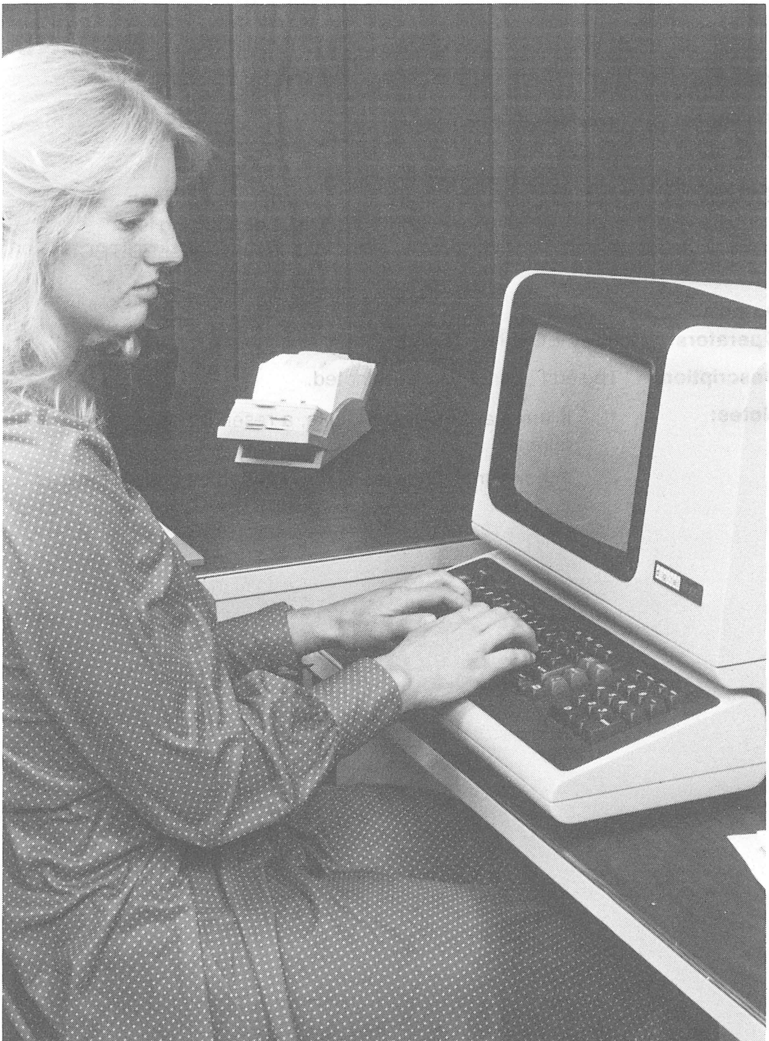
```
exit_flag ← true;              !terminate edit loop
                               !end processing is
                               !describe under EDITPC
                               instruction
```

Pattern Operators: 00 EO$END End Edit

Description: The edit operation is terminated.

Notes:

1. If there are still input digits, a reserved operand abort is taken.
2. If the source value is −0, the N condition code is cleared.



CHAPTER 17

PDP-11 COMPATIBILITY MODE

INTRODUCTION

In designing the VAX computer architecture, DIGITAL engineers were well aware of the need to establish a high level of compatibility with the large, well-established PDP-11 computer family. VAX represents the natural growth direction for many installations using PDP-11 machines and programs: to ease the growth, to quicken program transition, and to protect prior customer investment, it was important that VAXes display good compatibility features. Also, VAX had to provide compatibility to people who wanted to take advantage of its excellent program development tools in order to create and test programs that would later be loaded into PDP-11 computers.

The compatibility mode in the VAX architecture can make the computer "look like" a PDP-11 running the RSX-11M or IAS operating systems, naturally with some restrictions and requirements. A VAX computer treats compatibility mode programs like other processes, and can run them in its multiprogramming environment along with native mode programs. The computer should not be thought of as existing in one state or another, but rather as capable of handling both modes as the need arises.

So, if you are considering VAX for growth and for host program development, you will find that it provides useful compatibility with PDP-11s you already use or others you might be adding. And, of course, all the processors in the VAX family are 100% compatible with one another.

What follows in this chapter is a fairly detailed review of the powers and the restrictions of VAX compatibility mode. Naturally, if you need a greater depth of information, your DIGITAL Sales Representative or Software Specialist can supply it for you.

COMPATIBILITY MODE

VAX compatibility mode hardware, in conjunction with a compatibility mode software executive (which runs in native mode), can emulate the environment provided to user programs on a PDP-11. But this environment excludes from a complete PDP-11 the normal operation of the following features:

1. Privileged instructions such as HALT and RESET.
2. Special instructions such as traps and WAIT.
3. Access to internal processor registers (e.g., PSW and console switch register).

4. Direct access to trap and interrupt vectors.
5. Direct access to I/O devices. (Compatibility mode programs can directly reference I/O devices if and only if proper mapping has been established by native mode software.)
6. Interrupt servicing.
7. Stack overflow protection.
8. Alternate general register sets.
9. Any PDP-11 processor modes other than user mode, (i.e., kernel and supervisor) are not supported.
10. Floating point instructions.

Compatibility mode architecture is split into two parts. The first part is the PDP-11 environment provided by the VAX hardware. Details of the operation of PDP-11 compatible operations can be found in the appropriate PDP-11 Handbook. The second part is the hardware mechanisms provided in the VAX architecture that enable the implementation of various compatibility mode executives; this part is considered a subset of the VAX System Architecture.

COMPATIBILITY MODE USER ENVIRONMENT

General Registers And Address Modes

All of the PDP-11 general registers and addressing modes are provided in compatiblity mode. Side effects caused by a destination address calculation have no effect on source values (except in JSR), and autoincrement modes in JMP and JSR do not affect the new Program Counter. All addresses are 16 bits wide.

The Stack

General register R6 is used as the stack pointer by certain instructions, as in the PDP-11. It is not, however, used by the hardware for any exceptions or interrupts, nor is there any stack overflow protection in compatibility mode.

Processor Status Word

A subset of the full PDP-11 Processor Status Word is available in compatibility mode. The format of the compatibility mode PSW is:

The PSW can only be affected by the condition code instructions, RTI, and RTT. When an RTI or RTT instruction is executed, bits 15 through 5 in the saved PSW on the stack are ignored.

15 5	4	3	2	1	0
0	T	N	Z	V	C

Instructions

The following instructions are provided by the compatibility mode hardware.

Table 17-1 Compatibility Mode Instructions

Opcode (octal)	Mnemonic	Name
000002	RTI	Return from Interrupt
000006	RTT	Return from Trap
0001DD	JMP	Jump
00020R	RTS	Return from Subroutine
000240-000277	Condition Codes	
0003DD	SWAB	Swap Bytes
000400-003777	Branches	Branch
100000-103777	Branches	Branch
004RDD	JSR	Jump to Subroutine
.050DD	CLR(B)	Clear
.051DD	COM(B)	Complement
.052DD	INC(B)	Increment
.053DD	DEC(B)	Decrement
.054DD	NEG(B)	Negate
.055DD	ADC(B)	Add Carry
.056DD	SBC(B)	Subtract Carry
.057DD	TST(B)	Test
.060DD	ROR(B)	Rotate Right
.061DD	ROL(B)	Rotate Left
.062DD	ASR(B)	Arithmetic Shift Right
.063DD	ASL(B)	Arithmetic Shift Left
0065SS	MFPI*	Move from Previous Instruction Space

Opcode (octal)	**Mnemonic**	**Name**
0066DD	MTPI*	Move to Previous Instruction Space
1065SS	MFPD*	Move from Previous Data Space
1066DD	MTPD*	Move to Previous Data Space
0067DD	SXT	Sign Extend Word
070RSS	MUL	Multiply
071RSS	DIV	Divide
072RSS	ASH	Arithmetic Shift
073RSS	ASHC	Arithmetic Shift Combined
074RSS	XOR	Exclusive Or
077RNN	SOB	Subtract One and Branch
.1SSDD	MOV(B)	Move
.2SSSS	CMP(B)	Compare
.3SSSS	BIT(B)	Bit Test
.4SSDD	BIC(B)	Bit Clear
.5SSDD	BIS(B)	Bit Set
06SSDD	ADD	Add
16SSDD	SUB	Subtract

Legend:

R = Register specifier
SS = Source operand specifier
DD = Destination operand specifier
· = 0 for word operations; 1 for byte operations

* These instructions execute exactly as they would on a PDP-11 in user mode with Instruction and Data space overmapped. More specifically, they ignore the previous access level and act like PUSH and POP instructions referencing the current stack.

The following trap instructions cause the machine to enter native mode, where either the complete trap may be serviced, or the instruction may be simulated.

Table 17-2 Compatibility Mode Trap Instructions

Opcode (octal)	Mnemonic
000003	BPT
000004	IOT
104000-104377	EMT
104400-104777	TRAP

Some instructions, such as WAIT and RESET, are considered reserved instructions in compatibility mode. If encountered, they cause a fault to native mode. Table 17-3, following, lists such instructions. In addition, all other opcodes not defined above result in a fault to native mode.

Table 17-3 Compatibility Mode Reserved Instructions

Opcode (octal)	Mnemonic
000000	HALT
000001	WAIT
000005	RESET
000007	MFPT
00023N	SPL
0064NN	MARK
0070DD	CSM
07500R	FADD—FIS
07501R	FSUB—FIS
07502R	FMUL—FIS
07503R	FDIV—FIS
076XXX	Extended Instructions
1064SS	MTPS
1067DD	MFPS
17XXXX	FP11 Floating Point

Note that *no* floating point instructions are included in compatibility mode.

ENTERING AND LEAVING COMPATIBILITY MODE

Compatiblity mode is entered by executing an REI instruction with the compatibility mode bit set in the image of the PSL on the stack. Other bits in the PSL have the following effects:

N, Z, V, C	Condition Codes
T	T Bit
DV	Reserved operand fault if not zero
FU	Reserved operand fault if not zero
IV	Reserved operand fault if not zero
IPL	Reserved operand fault if not zero
PRV MOD	Reserved operand fault if not 3
CUR MOD	Reserved operand fault if not 3
IS	Reserved operand fault if not zero
FPD	Reserved operand fault if not zero
TP	T pending bit.

Native mode is reentered from compatibility mode by the compatibility mode program's causing an exception, or by an interrupt. The PSL pushed on the kernel or interrupt stack when leaving compatibility mode has all the bits that cause reserved operand faults in the above table set to the appropriate state.

Note that when an RTI or RTT instruction is executed in compatibility mode, the 11 high bits of the PSW are ignored. But when the PSW is restored as part of the PSL when going from native to compatibility mode, those bits must be zero or a reserved operand fault occurs.

General Register Usage

Compatibility mode registers 0 through 6 are bits 15 through 0 of VAX general registers 0 through 6, respectively. Compatibility mode register 7 (PC) is bits 15 through 0 of VAX general register 15 (PC). VAX registers 8 through 14 (SP) are not affected by compatibility mode. When entering compatibility mode, VAX register 7 and the upper halves of registers 0 through 6 and 15 are ignored. When an exception or interrupt occurs from compatibility mode, VAX register 7 is unpredictable and the upper halves of R0 through R6 and the stacked R15 (PC) are zero. Since there are no FP11 floating point instructions in compatibility mode, there are no floating accumulators.

COMPATIBILITY MODE MEMORY MANAGEMENT

The PDP-11 uses 16-bit byte addresses; hence compatibility mode programs are confined to execute in the first 64 Kbytes of the process part of virtual address space. There is a one-to-one correspondence between a compatibility mode virtual address and its VAX counterpart (e.g., virtual address 0 references the same location in both modes). A compatibility mode address is interpreted as follows:

31 16	15 9	8 0
0	PAGE	DISPLACEMENT

The PDP-11 capability of providing different access protection to different segments is provided in 8-block chunks, since protection is specified at the page level in the VAX architecture (i.e., one VAX page equals eight PDP-11 blocks).

The memory management system protects and relocates compatibility mode addresses in the normal manner. Thus, *all* of the memory management mechanisms available in native mode are available to the compatibility mode executive for managing both the virtual and physical memory of compatibility mode programs. All of the exception conditions that can be caused by memory management in native mode can also occur when relocating a compatibility mode address.

Most of the features of the PDP-11 memory management hardware affecting the user environment can be simulated with the VAX memory management system. Table 17-4 provides a general description of how this can be done; you may refer to the VAX Hardware Handbook and the appropriate PDP-11 Handbooks for details of each system.

Table 17-4

PDP-11 Memory Management Feature to be Simulated	**VAX Simulation Method**
Eight segments per user.	Eight segments can be simulated by dividing the 128 pages of the compatibility mode virtual address space into eight logical groups of 16 pages each, having possibly different protection.
Segment size from 64 bytes to 8 Kbytes (1 to 128 blocks) in 64-byte increments, using contiguous memory.	Segment size from 512 bytes to 8 Kbytes (1 to 16 pages) in 512-byte (1 page) increments, using discontiguous memory.
Forward growing segments (Expand Direction=0).	Can be simulated using page table entries specifying no access for those pages that are not allocated.
Backward growing segments (ED=1).	Can be simulated using page table entries specifying no access for those pages that are not allocated.

PDP-11 Memory Management Feature to be Simulated	**VAX Simulation Method**
Segments begin on any 64-byte boundary.	Segments begin on any 512-byte boundary.

Below is an example of how a PDP-11 environment can be created using the concepts in Table 17-4. Segments 0, 1, and 2 of the PDP-11 environment are program segments; 3 is unused; 4 and 5 are stack; and 6 and 7 are read/write data.

PDP-11 Environment				**VAX Page Table**	
Seg #	Size (bytes)	Expand Direction	Access	Page	Access
0	8K	Up	Read only	0-15	Read only
1	8K	Up	Read only	16-31	Read only
2	256	Up	Read only	32	Read only
3	0	—	None	33-77	No Access
4	1K	Down	Read/Write	78-79	Read/Write
5	8K	Down	Read/Write	80-95	Read/Write
6	8K	Up	Read/Write	96-111	Read/Write
7	2K	Up	Read/Write	112-115	Read/Write
				116-127	No Access

COMPATIBILITY MODE EXCEPTIONS AND INTERRUPTS

All interrupts and exception conditions which occur while the machine is in compatibility mode cause the machine to enter native mode (note that this includes backing up instruction side effects if necessary). The following exception conditions are specific to compatibility mode. All these exceptions create a three-longword frame on the kernel stack containing PSL, PC, and one longword of trap-specific information. Bits 15:0 of this longword contain a code indicating the specific type of trap and bits 31:16 are zero.

1. These are the opcodes that are defined in compatibility mode. The code for the reserved instruction trap is 0.
2. The code for the BPT instruction fault is 1.
3. The code for the IOT instruction fault is 2.
4. The fault code for the group of EMT instructions is 3.
5. The fault code for the group of TRAP instructions is 4.
6. Illegal instructions in compatibility mode are JMP and JSR instructions with a register destination. The fault code for illegal instructions is 5.

7. An odd address error abort is caused in compatibility mode whenever a word reference is attempted on a byte boundary. References that use the SP or PC are always word references, even if used in a byte instruction. The code for odd address errors is 6.

T BIT OPERATION IN COMPATIBILITY MODE

A compatibility mode trace fault occurs at the beginning of an instruction when the T bit is set in the PSW at the beginning of the prior instruction. On trace faults, a 2-longword kernel stack frame is created, containing the PSL and PC. IPL and IS are zero and CM is one in the stacked PSL. Compatibility mode trace faults use the same vector as native mode Trace fault. In fact, the rules for trace fault generation in compatibility mode are identical to those for native mode.

There are two ways to get the T bit set at the beginning of a compatibility mode instruction.

1. An RTT/RTI instruction is executed in compatibility mode and the T bit is set in the PSW image on the stack. In this case, the next instruction is executed (the one pointed to by the PC on the stack), and a trace fault is taken after that instruction.
2. An REI instruction is executed in native mode which has both the T bit and CM bit set (and T pending clear) in the saved PSL image on the stack. Again, one instruction is executed, and the T bit trap is taken. (The operations that occur as a function of these conditions are the same whether or not compatibility mode is being entered from the REI.)

The T bit interacts with other compatibility mode operations as follows:

1. T bit set at the beginning of a compatibility mode instruction which does not cause a compatibility mode fault.

   In this case, the instruction sets TP and executes. A trace fault is taken before the next instruction. The saved PSL has the T bit set and TP clear. The compatibility mode executive will do one of the following things:

   - If it services the exception directly, it may clear the T bit in the saved PSL on the kernel stack if it no longer wants to trace the program, or it may leave it set if it wants to continue tracing the program. It exits with an REI.
   - If it returns the trap to compatibility mode, it pushes a (16-bit) PC and (16-bit) PSW with the T bit set on the User stack to simulate the effect of the PDP-11 trace trap. It then clears the T bit in the saved PSL image on the kernel stack, changes the saved PC to point to the compatibility mode service routine,

and executes an REI. The compatibility mode service routine then may clear the T bit in the PSW image on its stack, if it does not want to continue tracing. The compatibility mode routine returns with RTT. (If it always clears the T bit in the saved PSW, it does not matter if it returns with RTI or RTT.)

2. T bit set at the beginning of an RTI or RTT.

   The RTT/RTI instruction executes and TP is set. A trace fault occurs before the next instruction is executed. There are two different cases, depending on whether or not the T bit was set in the image of the PSW which was popped from the stack by the instruction:

   - T bit not set.

     Neither TP nor T will be set in the saved PSL on the kernel stack.

   - T bit set.

     TP will not be set, and T will be set. This is the case for other compatibility mode instructions.

3. T bit set at the beginning of any instruction which causes a compatibility mode fault.

   The fault condition is serviced first. TP is clear and T is set in the saved PSL pushed on the kernel stack.

UNIMPLEMENTED PDP-11 TRAPS

Some traps that occur in PDP-11s that are not implemented in compatibility mode:

1. There is no stack overflow trap. Stack overflow can be provided by the compatibility mode executive using the memory management mechanisms.
2. There is no concept of a double error trap in compatibility mode, since the first error always puts the machine in native mode.
3. All other trap conditions such as power failure, memory parity, and memory management traps cause the machine to enter native mode.

COMPATIBILITY MODE I/O REFERENCES

Since I/O devices are accessible with all instructions in native mode (as in the PDP-11), I/O devices may be referenced directly from compatibility mode, if the memory mapping is set up to allow it. This may be done by mapping pages directly to I/O devices. Note that, in general, I/O devices will *not* appear in the physical address space on VAX machines in the same way that they do on PDP-11s, so existing PDP-11 programs that directly reference I/O devices probably will not work. In addition, compatibility mode programs can only do word or byte references; many VAX I/O devices may require that some references be 32 bits wide.

PROCESSOR REGISTERS

The only processor register available in compatibility mode is part of the PSW, and it may only be referenced with the condition code instructions, RTI, and RTT. Access to all other registers must be done in native mode.

PROGRAM SYNCHRONIZATION

All PDP-11s guarantee that read-modify-write operations to I/O device registers are interlocked; that is, the device can determine at the time of the read that the same register will be written as the next bus cycle. This synchronization also works in memory on most PDP-11s. In compatibility mode, instructions that have modify destinations will perform this synchronization for UNIBUS I/O device registers and never for memory.

CONCLUSION

As a powerful link joining the PDP-11 family and the VAX family, compatibility mode should help you expand your computing resources efficiently. And programs which, for one reason or another cannot take advantage of compatibility mode, usually can be fixed easily and quickly.

APPENDIXES

APPENDIX A

NOTATIONAL CONVENTIONS USED IN THIS HANDBOOK

Operational Notation

Graphical representations of memory, either physical or virtual, begin with low memory at the top of the diagram and progress downward toward higher addresses. This technique is illustrated in Figure A1

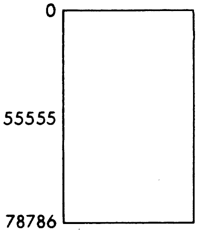


Figure A1 Memory Addressing Scheme

Unless otherwise noted, all numerical quantities are shown in decimal representation; decimal is the default radix of the system. Any other representation is shown using the radix of the number as a subscript:

$56A4C_{16}$

Operations notation uses an ALGOL-like format. For example, the ADWC instruction (Add With Carry) is represented as follows:

sum←sum+add+C

This shows the operation of adding the quantities "sum," "add," and "C" (for carry) and placing the result in "sum." Fuller details of this convention are given in Appendix E.

Range and Extent

An integer range is specified in English by the word "through," or in notational form by a double period "..", and is inclusive. For example, the range **0 through 4**, or **0..4**, means the integers 0,1,2,3 and 4.

An extent is given by a pair of numbers separated by a colon and is also inclusive. For example, **bits 7:3** specifies an extent of bits including bits 7,6,5,4, and 3.

Unpredictable and Undefined

Results specified as **unpredictable** may vary from moment to moment, implementation to implementation, and instruction to instruction within an implementation; engineering change orders (ECO's) may alter unpredictable results. Software should not depend on results specified as unpredictable.

Similarly, operations specified as **undefined** may vary from moment to moment, implementation to implementation, and instruction to instruction within an implementation. The operation can vary in effect from doing nothing up to stopping system operation. Of course nonprivileged software should avoid invoking undefined operations.

MBZ and Reserved

Fields identified with **MBZ** (Must Be Zero) should never be filled by software with a nonzero value. If the processor encounters a nonzero value in a field specified as MBZ, generally a reserved operand fault or abort occurs.

Certain fields and values accessible to privileged software are reserved to DIGITAL and the privileged software should not set nonzero or **reserved** values into these areas. (Fields reserved to DIGITAL and all MBZ fields may be used in the future to extend the standard architecture.)

In some cases, certain unassigned values are indicated as "reserved to CSS and customers." Only these values should be used for your nonstandard applications.

APPENDIX A1
DATA TABLES

INTRODUCTION

This appendix contains the following information:

- Hexadecimal-to-decimal conversion
- Decimal-to-hexadecimal conversion
- Hexadecimal addition
- Hexadecimal multiplication
- ASCII character set
- Hexadecimal-ASCII conversion
- Powers of 2
- Powers of 16

HEXADECIMAL-TO-DECIMAL CONVERSION

For each integer position of the hexadecimal value, locate the corresponding column integer in Table A-1 and record its decimal equivalent in the conversion table. Add the decimal equivalents to obtain the decimal value.

Example:

```
        D0500AD0(16)   =          ?(10)

            D0000000   =     3,489,660,928
              500000   =         5,242,880
                 A00   =             2,560
                  D0   =               208
  ------------------       -----------------
            D0500AD0   =     3,494,904,576
```

DECIMAL-TO-HEXADECIMAL CONVERSION

1. Locate in the conversion table (Table A-1) the largest decimal value that does not exceed the decimal number to converted.
2. Record the hexadecimal equivalent followed by the number of zeros (0) that corresponds to the integer column minus one.
3. Subtract the table decimal value from the decimal number to be converted.
4. Repeat steps 1-3 until the subtraction balance equals zero. Add the hexadecimal equivalents to obtain the hexadecimal value.

Example:

22,466 (10)	=	?(16)
22,466 −20,480	=	5000
1,986 −1,792	=	700
194 −192	=	C0
2 −2	=	2
22,466 (10)	=	57C2 (16)

HEXADECIMAL ADDITION

Table A-2 is a hexadecimal addition table for values from 0 through F. To add two hex numbers, locate one number in the left-hand column outside the body of the table and the other number in the topmost row above the body of the table. The intersection of these two numbers is the sum of the numbers. For example, to add A plus B, find A in the left column and B along the top row. The intersection of the two is 15.

HEXADECIMAL MULTIPLICATION

Table A-3 shows a hexadecimal multiplication table. To multiply two numbers, locate one in the left hand column outside the body of the table and the other in the topmost row outside the body of the table. The intersection of the two is the product of the two numbers. For example, to multiply 4 x A, locate 4 in the lefthand column and A in the topmost row. The intersection of the two is Z8, which is the product of the two numbers.

8		7		6		5		4		3		2		1	
HEX	DEC	HEX	DEC	HEX	DEC	HEX	DEC	HEX	DEC	HEX	DEC	HEX	DEC	HEX	DEC
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	268,435,456	1	16,777,216	1	1,048,576	1	65,536	1	4,096	1	256	1	16	1	1
2	536,870,912	2	33,554,432	2	2,097,152	2	131,072	2	8,192	2	512	2	32	2	2
3	805,306,368	3	50,331,648	3	3,145,728	3	196,608	3	12,288	3	768	3	48	3	3
4	1,073,741,824	4	67,108,864	4	4,194,304	4	262,144	4	16,384	4	1,024	4	64	4	4
5	1,342,177,280	5	83,886,080	5	5,242,880	5	327,680	5	20,480	5	1,280	5	80	5	5
6	1,610,612,736	6	100,663,296	6	6,291,456	6	393,216	6	24,576	6	1,536	6	96	6	6
7	1,897,048,192	7	117,440,512	7	7,340,032	7	458,752	7	28,672	7	1,792	7	112	7	7
8	2,147,483,643	8	134,217,728	8	8,388,608	8	524,288	8	32,768	8	2,048	8	128	8	8
9	2,415,919,104	9	150,994,944	9	9,437,184	9	589,824	9	36,864	9	2,304	9	144	9	9
A	2,684,354,560	A	167,772,160	A	10,485,760	A	655,360	A	40,960	A	2,560	A	160	A	10
B	2,952,790,016	B	184,549,376	B	11,534,336	B	720,896	B	45,056	B	2,816	B	176	B	11
C	3,221,225,472	C	201,326,592	C	12,582,912	C	786,432	C	49,152	C	3,072	C	192	C	12
D	3,489,660,928	D	218,103,808	D	13,631,488	D	851,968	D	53,248	D	3,328	D	208	D	13
E	3,758,096,384	E	234,881,024	E	14,680,064	E	917,504	E	57,344	E	3,584	E	224	E	14
F	4,026,531,840	F	251,658,240	F	15,728,640	F	983,040	F	61,440	F	3,840	F	240	F	15
BYTE				BYTE				BYTE				BYTE			
WORD								WORD							

Table A-2 HEXADECIMAL ADDITION

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F
1	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	10
2	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	10	11
3	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	10	11	12
4	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	10	11	12	13
5	05	06	07	08	09	0A	0B	0C	0D	0E	0F	10	11	12	13	14
6	06	07	08	09	0A	0B	0C	0D	0E	0F	10	11	12	13	14	15
7	07	08	09	0A	0B	0C	0D	0E	0F	10	11	12	13	14	15	16
8	08	09	0A	0B	0C	0D	0E	0F	10	11	12	13	14	15	16	17
9	09	0A	0B	0C	0D	0E	0F	10	11	12	13	14	15	16	17	18
A	0A	0B	0C	0D	0E	0F	10	11	12	13	14	15	16	17	18	19
B	0B	0C	0D	0E	0F	10	11	12	13	14	15	16	17	18	19	1A
C	0C	0D	0E	0F	10	11	12	13	14	15	16	17	18	19	1A	1B
D	0D	0E	0F	10	11	12	13	14	15	16	17	18	19	1A	1B	1C
E	0E	0F	10	11	12	13	14	15	16	17	18	19	1A	1B	1C	1D
F	0F	10	11	12	13	14	15	16	17	18	19	1A	1B	1C	1D	1E

Table A-3 HEXADECIMAL MULTIPLICATION

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
1	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F
2	00	02	04	06	08	0A	0C	0E	10	12	14	16	18	1A	1C	1E
3	00	03	06	09	0C	0F	12	15	18	1B	1E	21	24	27	2A	2D
4	00	04	08	0C	10	14	18	1C	20	24	28	2C	30	34	38	3C
5	00	05	0A	0F	14	19	1E	23	28	2D	32	37	3C	41	46	4B
6	00	06	0C	12	18	1E	24	2A	30	36	3C	42	48	4E	54	5A
7	00	07	0E	15	1C	23	2A	31	38	3F	46	4D	54	5B	62	29
8	00	08	10	18	20	28	30	38	40	48	50	50	60	68	70	78
9	00	09	12	1B	24	2D	36	3F	48	51	5A	63	6C	75	7E	87
A	00	0A	14	1E	28	32	3C	46	50	5A	64	6E	78	82	8C	96
B	00	0B	16	21	2C	37	42	4D	58	63	6E	79	84	8F	9A	A5
C	00	0C	18	24	30	3C	48	54	60	6C	78	84	90	9C	A8	B4
D	00	0D	1A	27	34	41	4E	5B	68	75	82	8F	9C	A9	B6	C3
E	00	0E	1C	2A	38	46	54	62	70	7E	8C	9A	A8	B6	C4	D2
F	00	0F	1E	2D	3C	4B	5A	69	78	87	96	A5	B4	C3	D2	E1

ASCII CHARACTER SET AND HEX-ASCII CONVERSION

Table A-4 represents the ASCII character set.

Table A-4 ASCII CHARACTER SET

	0	1	2	3	4	5	6	7
0	NUL	DLE	SP	0	@	P	‘	p
1	SOH	DC1	!	1	A	Q	a	q
2	STX	DC2	"	2	B	R	b	r
3	ETX	DC3	#	3	C	S	c	s
4	EOT	DC4	$	4	D	T	d	t
5	ENQ	NAK	%	5	E	U	e	u
6	ACK	SYN	&	6	F	V	f	v
7	BEL	ETB	’	7	G	W	g	w
8	BS	CAN	(	8	H	X	h	x
9	HT	EM	)	9	I	Y	i	y
A	LF	SUB	*	:	J	Z	j	z
B	VT	ESC	+	;	K	[	k	{
C	FF	FS	,	<	L	\	l	|
D	CR	GS	-	=	M	]	m	}
E	SO	RS	.	>	N	↑	n	~
F	SI	US	/	?	O	—	o	DEL

NUL	Null	DLE	Data Link Escape
SOH	Start of Heading	DC1	Device Control 1
STX	Start of Text	DC2	Device Control 2
ETX	End of Text	DC3	Device Control 3
EOT	End of Transmission	DC4	Device Control 4
ENQ	Enquiry	NAK	Negative Acknowledge
ACK	Acknowledge	SYN	Synchronous Idle
BEL	Bell	ETB	End of Transmission
Block			
BS	Backspace	CAN	Cancel
HT	Horizontal Tabulation	EM	End of Medium
LF	Line Feed	SUB	Substitute
VT	Vertical Tabulation	SC	ESCAPE
FF	Form Feed	FS	File Separator
CR	Carriage Return	GS	Group Separator
SO	Shift Out	RS	Record Separator
SI	Shift In	US	Unit Separator
SP	Space	DEL	Delete

POWERS OF 2 AND OF 16

For quick reference, the most commonly used powers of 2 and of 16 are shown below.

Powers of 2

2**n	n
256	8
512	9
1024	10
2048	11
4096	12
8192	13
16384	14
32768	15
65536	16
131072	17
262144	18
524288	19
1048576	20
2097152	21
4194304	22
8388608	23
16777216	24

Powers of 16

16**n	n
1	0
16	1
256	2
4096	3
65536	4
1048576	5
16777216	6
268435456	7
4294967296	8
68719476736	9
1099511627776	10
17592186044416	11
281474976710656	12
4503599627370496	13
72057594037927936	14
1152921504606846976	15

APPENDIX B
INSTRUCTION INDEX
By Mnemonic

MNEMONIC LISTING

MNEMONIC	INSTRUCTION	OPCODE	PAGE
ACBB	Add compare and branch byte	9D	268
ACBD	Add compare and branch D_floating	6F	268
ACBF	Add compare and branch F_floating	4F	268
ACBG	Add compare and branch G_floating	4FFD	268
ACBH	Add compare and branch H_floating	6FFD	268
ACBL	Add compare and branch longword	F1	268
ACBW	Add compare and branch word	3D	268
ADAWI	Add aligned word, interlocked	58	194
ADDB2	Add byte 2 operand	80	191
ADDB3	Add byte 3 operand	81	191
ADDD2	Add D_floating 2 operand	60	191
ADDD3	Add D_floating 3 operand	61	191
ADDF2	Add F_floating 2 operand	40	191
ADDF3	Add F_floating 3 operand	41	191
ADDG2	Add G_floating 2 operand	40FD	191
ADDG3	Add G_floating 3 operand	41FD	191
ADDH2	Add H_floating 2 operand	60FD	191
ADDH3	Add H_floating 3 operand	61FD	191
ADDL2	Add longword 2 operand	C0	191
ADDL3	Add longword 3 operand	C1	191
ADDP4	Add packed 4 operand	20	314
ADDP6	Add packed 6 operand	21	314
ADDW2	Add word 2 operand	A0	191
ADDW3	Add word 3 operand	A1	191
ADWC	Add with carry	D8	193
AOBLEQ	Add one and branch on less or equal	F3	270
AOBLSS	Add one and branch on less	F2	270
ASHL	Arithmetic shift longword	78	211
ASHP	Arithmetic shift and round packed	F8	330
ASHQ	Arithmetic shift quadword	79	211

MNEMONIC	INSTRUCTION	OPCODE	PAGE
BBC	Branch on bit clear	E1	264
BBCC	Branch on bit clear and clear	E5	265
BBCCI	Branch on bit clear and clear interlocked	E7	266
BBCS	Branch on bit clear and set	E3	265
BBS	Branch on bit set	E0	264
BBSC	Branch on bit set and clear	E4	265
BBSS	Branch on bit set and set	E2	265
BBSSI	Branch on bit set and set, interlocked	E6	266
BCC	Branch on carry clear	1E	261
BCS	Branch on carry set	1F	261
BEQL	Branch on equal (signed)	13	261
BEQLU	Branch on equal unsigned	13	261
BGEQ	Branch on greater or equal	18	261
BGEQU	Branch on greater or equal unsigned	1E	
BGTR	Branch on greater	14	261
BGTRU	Branch on greater unsigned	1A	261
BICB2	Bit clear byte 2 operand	8A	209
BICB3	Bit clear byte 3 operand	8B	209
BICL2	Bit clear longword 2 operand	CA	209
BICL3	Bit clear longword 3 operand	CB	209
BICPSW	Bit clear program status word	B9	223
BICW2	Bit clear word 2 operand	AA	209
BICW3	Bit clear word 3 operand	AB	209
BISB2	Bit set byte 2 operand	88	208
BISB3	Bit set byte 3 operand	89	208
BISL2	Bit set long 2 operand	C8	208
BISL3	Bit set long 3 operand	C9	208
BISPSW	Bit set program status word	B8	223
BISW2	Bit set word 2 operand	A8	208
BISW3	Bit set word 3 operand	A9	208
BITB	Bit test byte	93	207
BITL	Bit test longword	D3	207
BITW	Bit test word	B3	207
BLBC	Branch on low bit clear	E9	267
BLBS	Branch on low bit set	E8	267

MNEMONIC	INSTRUCTION	OPCODE	PAGE
BLEQ	Branch on less or equal	15	261
BLEQU	Branch on less or equal unsigned	1B	261
BLSS	Branch on less	19	261
BLSSU	Branch on less unsigned	1F	261
BNEQ	Branch on not equal	12	261
BNEQU	Branch on not equal unsigned	12	261
BPT	Break point fault	03	169
BRB	Branch with byte displacement	11	263
BRW	Branch with word displacement	31	263
BSBB	Branch to subroutine with byte displacement	10	275
BSBW	Branch to subroutine with word displacement	30	275
BUGL	Bugcheck longword	FDFF	170
BUGW	Bugcheck word	FEFF	170
BVC	Branch on overflow clear	1C	261
BVS	Branch on overflow set	1D	261
CALLG	Call with general argument list	FA	280
CALLS	Call with stack	FB	282
CASEB	Case byte	8F	273
CASEL	Case longword	CF	273
CASEW	Case word	AF	273
CHME	Change mode to executive	BD	158
CHMK	Change mode to kernel	BC	158
CHMS	Change mode to supervisor	BE	158
CHMU	Change mode to user	BF	158
CLRB	Clear byte	94	181
CLRD	Clear D_floating	7C	181
CLRF	Clear F_floating	D4	181
CLRG	Clear G_floating	7C	181
CLRH	Clear H_floating	7CFD	181
CLRL	Clear longword	D4	181
CLRO	Clear octaword	7CFD	181
CLRQ	Clear quadword	7C	181
CLRW	Clear word	B4	181
CMPB	Compare byte	91	188
CMPC3	Compare character 3 operand	29	294

MNEMONIC	INSTRUCTION	OPCODE	PAGE
CMPC5	Compare character 5 operand	2D	294
CMPD	Compare D_floating	71	188
CMPF	Compare F_floating	51	188
CMPG	Compare G_floating	51FD	188
CMPH	Compare H_floating	71FD	188
CMPL	Compare longword	D1	188
CMPP3	Compare packed 3 operand	35	313
CMPP4	Compare packed 4 operand	37	313
CMPV	Compare field	EC	255
CMPW	Compare word	B1	188
CMPZV	Compare zero-extended field	ED	255
CRC	Calculate cyclic redundancy check	0B	304
CVTBD	Convert byte to D_floating	6C	184
CVTBF	Convert byte to F_floating	4C	184
CVTBG	Convert byte to G_floating	4CFD	184
CVTBH	Convert byte to H_floating	6CFD	184
CVTBL	Convert byte to longword	98	184
CVTBW	Convert byte to word	99	184
CVTDB	Convert D_floating to byte	68	184
CVTDF	Convert D_floating to F_floating	76	184
CVTDH	Convert D_floating to H_floating	32FD	184
CVTDL	Convert D_floating to longword	6A	184
CVTDW	Convert D_floating to word	69	184
CVTFB	Convert F_floating to byte	48	184
CVTFD	Convert F_floating to D_floating	56	184
CVTFG	Convert F_floating to G_floating	99FD	184
CVTFH	Convert F_floating to H_floating	98FD	184
CVTFL	Convert F_floating to longword	4A	184
CVTFW	Convert F_floating to word	49	184
CVTGB	Convert G_floating to byte	48FD	184
CVTGF	Convert G_floating to F_floating	33FD	184
CVTGH	Convert G_floating to H_floating	56FD	184
CVTGL	Convert G_floating to longword	4AFD	184
CVTGW	Convert G_floating to word	49FD	184
CVTHB	Convert H_floating to byte	68FD	184
CVTHD	Convert H_floating to D_floating	F7FD	184
CVTHF	Convert H_floating to F_floating	F6FD	184
CVTHG	Convert H_floating to G_floating	76FD	184
CVTHL	Convert H_floating to longword	6AFD	184
CVTHW	Convert H_floating to word	69FD	184

MNEMONIC	INSTRUCTION	OPCODE	PAGE
CVTLB	Convert longword to byte	F6	184
CVTLD	Convert longword to D_floating	6E	184
CVTLF	Convert longword to F_floating	4E	184
CVTLG	Convert longword to G_floating	4EFD	184
CVTLH	Convert longword to H_floating	6EFD	184
CVTLP	Convert longword to packed	F9	321
CVTLW	Convert longword to word	F7	184
CVTPL	Convert packed to longword	36	322
CVTTP	Convert trailing numeric to packed	26	325
CVTPT	Convert packed to trailing numeric	24	323
CVTPS	Convert packed to leading separate numeric	08	327
CVTRDL	Convert rounded D_floating to longword	6B	184
CVTRFL	Convert rounded F_floating to longword	4B	184
CVTRGL	Convert rounded G_floating to longword	4BFD	184
CVTRHL	Convert rounded H_floating to longword	6BFD	184
CVTSP	Convert leading separate numeric to packed	09	329
CVTWB	Convert word to byte	33	184
CVTWD	Convert word to D_floating	6D	184
CVTWF	Convert word to F_floating	4D	184
CVTWG	Convert word to G_floating	4DFD	184
CVTWH	Convert word to H_floating	6DFD	184
CVTWL	Convert word to longword	32	184
DECB	Decrement byte	97	197
DECL	Decrement longword	D7	197
DECW	Decrement word	B7	197
DIVB2	Divide byte 2 operand	86	204
DIVB3	Divide byte 3 operand	87	204
DIVD2	Divide D_floating 2 operand	66	204
DIVD3	Divide D_floating 3 operand	67	204
DIVF2	Divide F_floating 2 operand	46	204

MNEMONIC	INSTRUCTION	OPCODE	PAGE
DIVF3	Divide F_floating 3 operand	47	204
DIVG2	Divide G_floating 2 operand	46FD	204
DIVG3	Divide G_floating 3 operand	47FD	204
DIVH2	Divide H_floating 2 operand	66FD	204
DIVH3	Divide H_floating 3 operand	67FD	204
DIVL2	Divide longword 2 operand	C6	204
DIVL3	Divide longword 3 operand	C7	204
DIVP	Divide packed	27	319
DIVW2	Divide word 2 operand	A6	204
DIVW3	Divide word 3 operand	A7	204
EDITPC	Edit packed to character	38	335
EDIV	Extended divide	7B	206
EMODD	Extended modulus D_floating	74	202
EMODF	Extended modulus F_floating	54	202
EMODG	Extended modulus G_floating	54FD	202
EMODH	Extended modulus H_floating	74FD	202
EMUL	Extended multiply	7A	201
EXTV	Extract field	EE	253
EXTZV	Extract zero-extended field	EF	253
FFC	Find first clear bit	EB	251
FFS	Find first set bit	EA	251
HALT	Halt	00	171
INCB	Increment byte	96	189
INCL	Increment longword	D6	189
INCW	Increment word	B6	189
INDEX	Compute index	0A	226
INSQHI	Insert into queue head, interlocked	5C	240
INSQTI	Insert into queue tail, interlocked	5D	240
INSQUE	Insert into queue	0E	232
INSV	Insert field	F0	257
JMP	Jump	17	263
JSB	Jump to subroutine	16	275
LDPCTX	Load process context	06	163
LOCC	Locate character	3A	299

MNEMONIC	INSTRUCTION	OPCODE	PAGE
MATCHC	Match characters	39	301
MCOMB	Move complemented byte	92	183
MCOML	Move complemented long	D2	183
MCOMW	Move complemented word	B2	183
MFPR	Move from privilege register	DB	165
MNEGB	Move negated byte	8E	187
MNEGD	Move negated D_floating	72	182
MNEGF	Move negated F_floating	52	182
MNEGG	Move Negated G_floating	52FD	182
MNEGH	Move Negated H_floating	72FD	182
MNEGL	Move negated longword	CE	182
MNEGW	Move negated word	AE	182
MOVAB	Move address of byte	9E	224
MOVAD	Move address of D_floating	7E	224
MOVAF	Move address of F_floating	DE	224
MOVAG	Move Address of G_floating	7E	224
MOVAH	Move Address of H_floating	7EFD	224
MOVAL	Move address of longword	DE	224
MOVAO	Move Address of octaword	7EFD	224
MOVAQ	Move address of quadword	7E	224
MOVAW	Move address of word	3E	224
MOVB	Move byte	90	179
MOVC3	Move character 3 operand	28	289
MOVC5	Move character 5 operand	2C	289
MOVD	Move D_floating	70	179
MOVF	Move F_floating	50	179
MOVG	Move G_floating	50FD	179
MOVH	Move H_floating	70FD	179
MOVL	Move longword	D0	179
MOVO	Move octaword	7DFD	179
MOVP	Move packed	34	312
MOVPSL	Move processor status longword	DC	222
MOVQ	Move quadword	7D	179
MOVTC	Move translated characters	2E	290
MOVTUC	Move translated until character	2F	292
MOVW	Move word	B0	179
MOVZBL	Move zero-extended byte to longword	9A	187

MNEMONIC	INSTRUCTION	OPCODE	PAGE
MOVZBW	Move zero-extended byte to word	9B	187
MOVZWL	Move zero-extended word to longword	3C	187
MTPR	Move to privilege register	DA	165
MULB2	Multiply byte 2 operand	84	199
MULB3	Multiply byte 3 operand	85	199
MULD2	Multiply D_floating 2 operand	64	199
MULD3	Multiply D_floating 3 operand	65	199
MULF2	Multiply F_floating 2 operand	44	199
MULF3	Multiply F_floating 3 operand	45	199
MULG2	Multiply G_floating 2 operand	44FD	199
MULG3	Multiply G_floating 3 operand	45FD	199
MULH2	Multiply H_floating 2 operand	64FD	199
MULH3	Multiply H_floating 3 operand	65FD	199
MULL2	Multiply longword 2 operand	C4	199
MULL3	Multiply longword 3 operand	C5	199
MULP	Multiply packed	25	318
MULW2	Multiply word 2 operand	A4	199
MULW3	Multiply word 3 operand	A5	199
NOP	No operation	01	
POLYD	Evaluate polynomial D_floating	75	214
POLYF	Evaluate polynomial F_floating	55	214
POLYG	Evaluate polynomial G_floating	55FD	214
POLYH	Evaluate polynomial H_floating	75FD	214
POPR	Pop registers	BA	221
PROBER	Probe read access	0C	160
PROBEW	Probe write access	0D	160
PUSHAB	Push address byte	9F	224
PUSHAD	Push address of D_floating	7F	224
PUSHAF	Push address of F_floating	DF	224
PUSHAG	Push Address of G_floating	7F	224
PUSHAH	Push Address of H_floating	7FFD	224
PUSHAL	Push address of longword	DF	224
PUSHAO	Push address of octaword	7FFD	224
PUSHAQ	Push address of quadword	7F	224
PUSHAW	Push address of word	3F	224
PUSHL	Push longword	DD	180
PUSHR	Push registers	BB	220

MNEMONIC	INSTRUCTION	OPCODE	PAGE
REI	Return from exception or interrupt	02	161
REMQHI	Remove from queue head, interlocked	5E	245
REMQTI	Remove from queue tail, interlocked	5F	248
REMQUE	Remove from queue	0F	234
RET	Return from called procedure	04	284
ROTL	Rotate longword	9C	212
RSB	Return from subroutine	05	276
SBWC	Subtract with carry	D9	198
SCANC	Scan for character	2A	297
SKPC	Skip character	3B	299
SOBGEQ	Subtract one and branch on greater or equal	F4	271
SOBGTR	Subtract one and branch on greater	F5	271
SPANC	Span characters	2B	297
SUBB2	Subtract byte 2 operand	82	195
SUBB3	Subtract byte 3 operand	83	195
SUBD2	Subtract D_floating 2 operand	62	195
SUBD3	Subtract D_floating 3 operand	63	195
SUBF2	Subtract F_floating 2 operand	42	195
SUBF3	Subtract F_floating 3 operand	43	195
SUBG2	Subtract G_floating 2 operand	42FD	195
SUBG3	Subtract G_floating 3 operand	43FD	195
SUBH2	Subtract H_floating 2 operand	62FD	195
SUBH3	Subtract H_floating 3 operand	63FD	195
SUBL2	Subtract longword 2 operand	C2	195
SUBL3	Subtract longword 3 operand	C3	195
SUBP4	Subtract packed 4 operand	22	316
SUBP6	Subtract packed 6 operand	23	316
SUBW2	Subtract word 2 operand	A2	195
SUBW3	Subtract word 3 operand	A3	195
SVPCTX	Save process context	07	163
TSTB	Test byte	95	190
TSTD	Test D_floating	73	190
TSTF	Test F_floating	53	190
TSTG	Test G_floating	53FD	190
TSTH	Test H_floating	73FD	190
TSTL	Test long	D5	190
TSTW	Test word	B5	190

MNEMONIC	INSTRUCTION	OPCODE	PAGE
XFC	Extended function call	FC	168
XORB2	Exclusive OR byte 2 operand	8C	210
XORB3	Exclusive OR byte 3 operand	8D	210
XORL2	Exclusive OR longword 2 operand	CC	210
XORL3	Exclusive OR longword 3 operand	CD	210
XORW2	Exclusive OR word 2 operand	TC	210
XORW3	Exclusive OR word 3 operand	AD	210
ESCD	Reserved to DIGITAL	FD	
ESCE	Reserved to DIGITAL	FE	
ESCF	Reserved to DIGITAL	FF	

Reserved to DIGITAL		57;59;5A;5B;77;
		00FD to 31FD;
		34FD to 3FFD;
		57FD, 58FD,
		...5FFD;
		77FD,78FD,
		...7FFD;
		80FD to 97FD;
		9AFD to F5FD;
		F8FD to FCFF.	

APPENDIX B1

INSTRUCTION INDEX BY OPCODE

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
OA	INDEX	Compute index
OB	CRC	Calculate cyclic redundancy check
OC	PROBER	Probe read access
OD	PROBEW	Prove write access
OE	INSQUE	Insert into queue
OF	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, Branch on not equal unsigned
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

OPCODE	MNEMONIC	INSTRUCTION
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
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 longword
33	CVTWB	Convert word to byte
34	MOVP	Move packed
35	CMPP3	Compare packed 3 operand
36	CVTPL	Convert packed to longword
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 longword

OPCODE	MNEMONIC	INSTRUCTION
3D	ACBW	Add compare and branch word
3E	MOVAW	Move address of word
3F	PUSHAW	Push address of word
40	ADDF2	Add F_floating 2 operand
41	ADDF3	Add F_floating 3 operand
42	SUBF2	Subtract F_floating 2 operand
43	SUBF3	Subtract F_floating 3 operand
44	MULF2	Multiply F_floating 2 operand
45	MULF3	Multiply F_floating 3 operand
46	DIVF2	Divide F_floating 2 operand
47	DIVF3	Divide F_floating 3 operand
48	CVTFB	Convert F_floating to byte
49	CVTFW	Convert F_floating to word
4A	CVTFL	Convert F_floating to longword
4B	CVTRFL	Convert rounded F_floating to longword
4C	CVTBF	Convert byte to F_floating
4D	CVTWF	Convert word to F_floating
4E	CVTLF	Convert longword to F_floating
4F	ACBF	Add compare and branch floating
50	MOVF	Move F_floating
51	CMPF	Compare F_floating
52	MNEGF	Move negated F_floating
53	TSTF	Test F_floating
54	EMODF	Extended modulus F_floating
55	POLYF	Evaluate polynomial F_floating
56	CVTFD	Convert F_floating to D_floating
57		RESERVED to DIGITAL
58	ADAWI	Add aligned word, interlocked
59		RESERVED to DIGITAL
5A		RESERVED to DIGITAL
5B		RESERVED to DIGITAL
5C	INSQHI	Insert into queue head, interlocked
5D	INSQTI	Insert into queue tail, interlocked
5E	REMQHI	Remove from queue head, interlocked

OPCODE	MNEMONIC	INSTRUCTION
5F	REMQTI	Remove from queue tail, interlocked
60	ADDD2	Add D_floating 2 operand
61	ADDD3	Add D_floating 3 operand
62	SUBD2	Subtract D_floating 2 operand
63	SUBD3	Subtract D_floating 3 operand
64	MULD2	Multiply D_floating 2 operand
65	MULD3	Multiply D_floating 3 operand
66	DIVD2	Divide D_floating 2 operand
67	DIVD3	Divide D_floating 3 operand
68	CVTDB	Convert D_floating to byte
69	CVTDW	Convert D_floating to word
6A	CVTDL	Convert D_floating to longword
6B	CVTRDL	Convert rounded D_floating to longword
6C	CVTBD	Convert byte to D_floating
6D	CVTWD	Convert word to D_floating
6E	CVTLD	Convert longword to D_floating
6F	ACBD	Add compare and branch D_floating
70	MOVD	Move D_floating
71	CMPD	Compare D_floating
72	MNEGD	Move negated D_floating
73	TSTD	Test D_floating
74	EMODD	Extended modulus D_floating
75	POLYD	Evaluate polynomial D_floating
76	CVTDF	Convert D_floating to F_floating
77		RESERVED to DIGITAL
78	ASHL	Arithmetic shift longword
79	ASHQ	Arithmetic shift quadword
7A	EMUL	Extended multiply
7B	EDIV	Extended divide
7C	CLRQ, CLRD, CLRG	Clear quadword, Clear D_floating, Clear G_floating
7D	MOVQ	Move quadword
7E	MOVAQ, MOVAD, MOVAG	Move address of quadword, Move address of D_floating, Move address of G_floating

OPCODE	MNEMONIC	INSTRUCTION
7F	PUSHAQ, PUSHAD, PUSHAG	Push address of quadword, Push address of D_floating, Push address of G_floating
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 longword
99	CVTBW	Convert byte to word
9A	MOVZBL	Move zero-extended byte to longword
9B	MOVZBW	Move zero-extended byte to word
9C	ROTL	Rotate longword
9D	ACBB	Add compare and branch byte
9E	MOVAB	Move address of byte
9F	PUSHAB	Push address of byte

OPCODE	MNEMONIC	INSTRUCTION
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
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 register
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 longword 2 operand
C1	ADDL3	Add longword 3 operand
C2	SUBL2	Subtract longword 2 operand

OPCODE	MNEMONIC	INSTRUCTION
C3	SUBL3	Subtract longword 3 operand
C4	MULL2	Multiply longword 2 operand
C5	MULL3	Multiply longword 3 operand
C6	DIVL2	Divide longword 2 operand
C7	DIVL3	Divide longword 3 operand
C8	BISL2	Bit set longword 2 operand
C9	BISL3	Bit set longword 3 operand
CA	BICL2	Bit clear longword 2 operand
CB	BICL3	Bit clear longword 3 operand
CC	XORL2	Exclusive OR longword 2 operand
CD	XORL3	Exclusive OR longword 3 operand
CE	MNEGL	Move negated longword
CF	CASEL	Case longword
D0	MOVL	Move longword
D1	CMPL	Compare longword
D2	MCOML	Move complemented longword
D3	BITL	Bit test longword
D4	CLRL, CLRF	Clear longword, Clear F_floating
D5	TSTL	Test longword
D6	INCL	Increment longword
D7	DECL	Decrement longword
D8	ADWC	Add with carry
D9	SBWC	Subtract with carry
DA	MTPR	Move to processor register
DB	MFPR	Move from processor register
DC	MOVPSL	Move processor status longword
DD	PUSHL	Push longword
DE	MOVAL, MOVAF	Move address of longword, Move address of F_floating
DF	PUSHAL, PUSHAF	Push address of longword, Push address of F_floating
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

OPCODE	MNEMONIC	INSTRUCTION
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 longword
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 longword to byte
F7	CVTLW	Convert longword to word
F8	ASHP	Arithmetic shift and round packed
F9	CVTLP	Convert longword to packed
FA	CALLG	Call with general argument list
FB	CALLS	Call with stack argument list
FC	XFC	Extended function call
FD	ESCD to DIGITAL	
FE	ESCE to DIGITAL	
FF	ESCF to DIGITAL	

TWO-BYTE OPCODES

OPCODE	MNEMONIC	INSTRUCTION
00FD		
to		
31FD	RESERVED TO DEC	
32FD	CVTDH	Convert D_floating to H_floating
33FD	CVTGF	Convert G_floating to F_floating
34FD		
to		
3FFD	RESERVED to DEC	
40FD	ADDG2	Add G_floating 2 operand
41FD	ADDG3	Add G_floating 3 operand
42FD	SUBG2	Subtract G_floating 2 operand
43FD	SUBG3	Subtract G_floating 3 operand
44FD	MULG2	Multiply G_floating 2 operand
45FD	MULG3	Multiply G_floating 3 operand
46FD	DIVG2	Divide G_floating 2 operand
47FD	DIVG3	Divide G_floating 3 operand
48FD	CVTGB	Convert G_floating to byte
49FD	CVTGW	Convert G_floating to word
4AFD	CVTGL	Convert G_floating to longword
4BFD	CVTRGL	Convert rounded G_floating to longword
4CFD	CVTBG	Convert byte to G_floating
4DFD	CVTWG	Convert word to G_floating
4EFD	CVTLG	Convert longword to G_floating
4FFD	ACBG	Add, Compare and Branch G_floating
50FD	MOVG	Move G_floating
51FD	CMPG	Compare G_floating
52FD	MNEGG	Move negated G_floating
53FD	TSTG	Test G_floating
54FD	EMODG	Extended modulus G_floating
55FD	POLYG	Polynomial Evaluation G_floating
56FD	CVTGH	Convert G_floating to H_floating
57FD	RESERVED to DEC	
58FD	RESERVED to DEC	
59FD	RESERVED to DEC	

OPCODE	MNEMONIC	INSTRUCTION
5AFD	RESERVED to DEC	
5BFD	RESERVED to DEC	
5CFD	RESERVED to DEC	
5DFD	RESERVED to DEC	
5EFD	RESERVED to DEC	
5FFD	RESERVED to DEC	
60FD	ADDH2	Add H_floating 2 operand
61FD	ADDH3	Add H_floating 3 operand
62FD	SUBH2	Subtract H_floating 2 operand
63FD	SUBH3	Subtract H_floating 3 operand
64FD	MULH2	Multiply H_floating 2 operand
65FD	MULH3	Multiply H_floating 3 operand
66FD	DIVH2	Divide H_floating 2 operand
67FD	DIVH3	Divide H_floating 3 operand
68FD	CVTHB	Convert H_floating to byte
69FD	CVTHW	Convert H_floating to word
6AFD	CVTHL	Convert H_floating to longword
6BFD	CVTRHL	Convert rounded H_floating to longword
6CFD	CVTBH	Convert byte to H_floating
6DFD	CVTWH	Convert word to H_floating
6EFD	CVTLH	Convert longword to H_floating
6FFD	ACBH	Add, Compare and Branch H_floating
70FD	MOVH	Move H_floating
71FD	CMPH	Compare H_floating
72FD	MNEGH	Move negated H_floating
73FD	TSTH	Test H_floating
74FD	EMODH	Extended modulus H_floating
75FD	POLYH	Polynomial evaluation H_floating
76FD	CVTHG	Convert H_floating to G_floating
77FD	RESERVED to DEC	
78FD	RESERVED to DEC	
79FD	RESERVED to DEC	
7AFD	RESERVED to DEC	
7BFD	RESERVED to DEC	
7CFD	CLRH,CLRO	Clear H_floating, Clear octaword
7DFD	MOVO	Move octaword

OPCODE	MNEMONIC	INSTRUCTION
7EFD	MOVAH,MOVAO	Move address of H_floating, Move address of octaword
7FFD	PUSHAH,PUSHAO	Push address of H_floating, Push address of octaword
80FD to 97FD	RESERVED to DIGITAL	
98FD	CVTFH	Convert F_floating to H_floating
99FD	CVTFG	Convert F_floating to G_floating
9AFD to F5FD	RESERVED to DIGITAL	
F6FD	CVTHF	Convert H_floating to F_floating
F7FD	CVTHD	Convert H_floating to D_floating
F8FD to FCFF	RESERVED to DIGITAL	
FDFF	BUGL	BUGCHECK longword
FEFF	BUGW	BUGCHECK word
FFFF	RESERVED for all time	

APPENDIX C
VAX PROCEDURE CALLING AND CONDITION HANDLING STANDARD

Version 8.0, October, 1980

This appendix is the VAX Procedure Calling Standard used with the VAX hardware procedure call mechanism. This standard applies to:

1. All externally callable interfaces in DIGITAL-supported, standard system software
2. All intermodule CALLs to major VAX components
3. All external procedure CALLs generated by standard DIGITAL language processors

This standard does not apply to calls to internal (local) routines, or language support routines. Within a single module, the language processor or programmer can use a variety of other linkage and argument-passing techniques.

The standard defines mechanisms for passing arguments by immediate value, by reference, and by descriptor. However, the immediate value mechanism is intended for use only by VAX/VMS system services and within programs written in BLISS or MACRO.

The procedure CALL mechanism depends on agreement between the calling and called procedures to interpret the argument list. The argument list does not fully describe itself. This standard requires language extensions to permit a calling program to generate some of the argument passing mechanisms expected by called procedures.

This standard specifies the following attributes of the interfaces between modules:

- Calling sequence—the instructions at the call site and at the entry point
- Argument list—the structure of the list describing the arguments to the called procedure
- Function value return—the form and conventions for the return of the function value as a value or as a condition value to indicate success or failure
- Register usage—which registers are preserved and who is responsible for preserving them
- Stack usage—rules governing the use of the stack
- Argument data types—the data types of arguments that can be passed

- Argument descriptor formats—how descriptors are passed for the more complex arguments
- Condition handling—how exception conditions are signaled and how they can be handled in a modular fashion
- Stack unwinding—how the current thread of execution can be aborted cleanly

The goals in developing the VAX Procedure Calling Standard were:

- The standard must be applicable to all intermodule callable interfaces in the VAX software system. Specifically, the standard must consider the requirements of MACRO, BLISS, BASIC, CORAL, FORTRAN, PASCAL, PL/I, COBOL and CALLs to the operating system and library procedures. The needs of other languages that DIGITAL may support in the future must be met by the standard or by compatible revision to it.
- The standard should not include capabilities for lower level components (such as BLISS, MACRO, operating system) that cannot be invoked from the higher level languages.
- The calling program and procedure can be written in different languages. The standard attempts to reduce the need for use of language extensions for mixed language programs.
- The procedure mechanism must be sufficiently economical in both space and time to be used and usable as the only calling mechanism within VAX.
- The standard should contribute to the writing of error-free, modular, and maintainable software. Effective sharing and reuse of VAX software modules are significant goals.
- The standard must allow the called procedure a variety of techniques for argument handling. The called procedure can:
  1. Reference arguments indirectly through the argument list
  2. Copy atomic data types, strings and arrays
  3. Copy addresses of atomic data types, strings and arrays
- The standard should provide the programmer with some control over fixing, reporting, and flow of control on hardware and software exceptions.
- The standard should provide subsystem and application writers with the ability to override system messages to provide a more suitable application oriented interface.
- The standard should add no space or time overhead to procedure calls and returns that do not establish handlers and should minimize time overhead for establishing handlers at the cost of increased time overhead when exceptions occur.

Some possible attributes of a procedure-calling mechanism were considered and rejected. Specific nongoals for the VAX procedure CALL mechanism include:

- It is not necessary for the procedure mechanism to provide complete checking of argument data types, data structures, and parameter access. The VAX protection and memory-management system is not dependent upon "correct" interactions between user-level calling and called procedures. Such extended checking may be desirable in some circumstances, but system integrity is not dependent upon it.
- The VAX procedure mechanism need not provide complete information for an interpretive DEBUG facility. The definition of the DEBUG facility includes a DEBUG symbol table which contains the required descriptive information.

The following definitions apply to this standard:

- A **procedure** is a closed sequence of instructions that is entered from and returns control to the calling program.
- A **function** is a procedure that returns a single value according to the standard conventions for value returning. If additional values are returned, they are returned via the argument list.
- A **subroutine** is a procedure that does not return a known value according to the standard conventions for value returning. If values are returned, they are returned via the argument list.
- An **address** is a 32-bit VAX address positioned in a longword item.
- An **argument list** is a vector of longwords that represents a procedure parameter list and possibly a function value.
- **Immediate value** is a mechanism for passing input parameters in which the actual value is provided in the longword argument list entry by the calling program.
- **Reference** is a mechanism for passing parameters in which the address of the parameters is provided in the longword argument list by the calling program.
- **Descriptor** is a mechanism for passing parameters in which the address of a descriptor is provided in the longword argument list entry. The descriptor contains the address of the parameter, the data type, size and additional information needed to describe fully the data passed.
- An **exception condition** is a hardware or software detected event that alters the normal flow of instruction execution. It usually indicates a failure.
- A **condition value** is a 32-bit value used to identify an exception

condition uniquely. A condition value may be returned to a calling program as a function value or signaled using the VAX signaling mechanism.

- **Language support procedures** are called implicitly to implement higher level language constructs. They are not intended to be called explicitly from user programs.
- **Library procedures** are called explicitly using the equivalent of a CALL statement or function reference. They are usually language-independent.

1. CALLING SEQUENCE

At the option of the calling program, the called procedure is invoked using either the CALLG or CALLS instruction:

```
        CALLG       arglst, proc
        CALLS       argcnt, proc
```

CALLS pushes the argument count argcnt onto the stack, as a longword and sets the argument pointer, AP, to the top of the stack. The complete sequence using CALLS is:

```
        push       argn
        ...
        push       arg1
        CALLS        #n, proc
```

If the called procedure returns control to the calling program, control must return to the instruction immediately following the CALLG or CALLS instruction. Skip returns and GOTO returns are only allowed during stack unwind operations.

The called procedure returns control to the calling program by executing the return instruction, RET.

2. ARGUMENT LIST

The argument list is the primary means of passing information to and receiving results from a procedure.

2.1 Argument List Format

The argument list is a sequence of longwords:

The first longword is always present and contains the argument count as an unsigned integer in the low byte. The 24 high-order bits are reserved to DIGITAL and must be zero. To access the argument count, the called procedure must ignore the reserved bits and access the count as an unsigned byte (for example MOVZBL, TSTB, or CMPB).

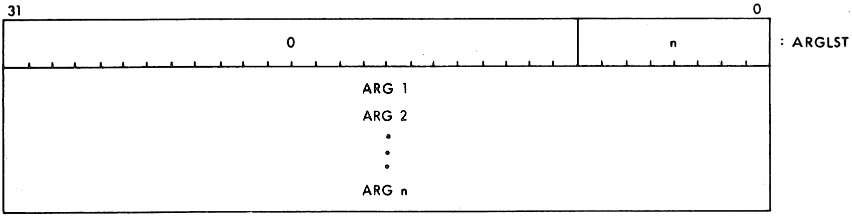


Figure C-1 Argument List

The remaining longwords can be:

1. An uninterpreted 32-bit value (immediate value mechanism). If the called procedure expects fewer than 32 bits, it accesses the low-order bits and ignores the high-order bits.
2. An address (reference mechanism). It is typically a pointer to a scalar data item, an array, a structure, a record, or a procedure.
3. An address of a descriptor (descriptor mechanism). See Section 8 for descriptor formats.

The standard permits immediate value, reference, descriptor, or combinations of these mechanisms. Interpretation of each argument list entry depends on agreement between the calling and called procedures. Higher level languages use the reference or decriptor mechanisms for passing input parameters. VAX/VMS System Services and MACRO or BLISS programs use all three mechanisms.

A procedure with no arguments is called with a list consisting of a 0 argument count longword. This is accomplished as follows:

```
        CALLS      #0, proc
```

A missing or null argument, for example CALL SUB(A,,B), is represented by an argument list entry consisting of a longword 0. Some procedures allow trailing null arguments to be omitted, others require all arguments. See each procedure's specification for details.

The argument list must be treated as read-only data by the called procedure and may be allocated in read-only memory at the option of the calling program.

2.2 Argument Lists and Higher Level Languages

Higher level language functional notations for procedure calls are mapped into VAX argument lists according to the folowing rules:

1. Arguments are mapped from left to right to increasing argument list offsets. The leftmost (first) argument has an address of arglst+4, the next has an address of arglst+8, etc. (or arglst+8, arglst+12, etc. when the contents of arglst+4 specify where the function value is to be returned—see Section 3).
2. Each argument position corresponds to a single VAX argument list entry.

2.2.1 Order of Argument Evaluation — Since most higher level languages do not specify the order of evaluation (with respect to side effects) of arguments, those language processors can evaluate arguments in any convenient order.

In constructing an argument list on the stack, a language processor can evaluate arguments from right to left and push their values on the stack. If call-by-reference semantics are used, argument expressions can be evaluated from left to right, with pointers to the expression values or descriptors being pushed from right to left.

The choice of argument evaluation order and code generation strategy is constrained only by the definition of the particular language. Programs should not be written that depend on the order of evaluation of arguments.

2.2.2 Language Extensions for Argument Transmission — The VAX procedure standard permits arguments to be passed by immediate value, by reference, or by descriptor. All language processors, except MACRO and BLISS, pass arguments by reference or descriptor by default.

Language extensions are needed to reconcile the different argument passing mechanisms. In addition to the default passing mechanism used, each language processor is required to give the user explicit control of the argument passing mechanism in the calling program for the data types supported by the language as follows:

Data Type	Section	Immediate Value	Reference	Descriptor (at least one)
Atomic ≤ 32 bits	7.1	Yes	Yes	Yes
Atomic > 32 bits	7.1	No	Yes	Yes
String	7.2	No	Yes	Yes
Miscellaneous	7.3	No*	No	No
Array	8	No	Yes	Yes

* For those languages supporting the Bound Procedure Value data type, a language extension is required to pass it by immediate value in order to be able to interface with VMS system services and other software. See Section 7.3.

For example, FORTRAN provides the following intrinsic compile-time functions:

%VAL(arg)	Immediate Value Mechanism—Corresponding argument list entry is the 32-bit value of the argument, arg, as defined in the language.
%REF(arg)	Reference Mechanism—Corresponding argument list entry contains the address of the value of the argument, arg, as defined in the language.
%DESCR(arg)	Descriptor Mechanism—Corresponding argument list entry contains the address of a VAX descriptor of the argument, arg, as defined in Section 8.

These intrinsic funtions can be used in the syntax of a procedure call to control generation of the argument list. For example:

```
CALL SUB1(%VAL(123), %REF(X), %DESCR(A))
```

In other languages the same effect might be achieved by appropriate attributes of the declaration of SUB1 made in the calling program. Thus, the user might write:

```
CALL SUB1 (123, X, A)
```

after making the external declaration for SUB1.

3. FUNCTION VALUE RETURN

A function value is returned in register R0 if its data type is representable in 32 bits or registers R0 and R1 if representable in 64 bits. Two separate 32-bit entities cannot be returned in R0 and R1 because higher level languages cannot process them.

1. If the maximum length of the function value is known (for example, octaword integer, H_floating, or fixed-length string), the

calling program can allocate the required storage and pass the address of the storage or a decriptor for the storage as the first argument.

2. If the maximum length of a string function value is not known to the calling program, the calling program can allocate a dynamic string descriptor. The called procedure then allocates storage for the function value and updates the contents of the dynamic string descriptor using VAX Run Time Library procedures. See Section 8.3.

Some procedures, such as operating system calls and many library procedures, return a success/fail value as a longword function value in R0. Bit 0 of the value is set (Boolean true) for a success and clear (Boolean false) for a failure. The particular success or failure status is encoded in the remaining 31 bits, as described in Section 4.

4. CONDITION VALUE

VAX uses condition values for the following:

- To indicate the success or failure of a called procedure as a function value
- To describe an exception condition when an exception is signaled
- To identify system messages
- To report program success or failure to the command language level

A condition value is a longword that includes fields to describe the software component generating the value, the reason the value was generated and the error severity status. The format of the condition value is:

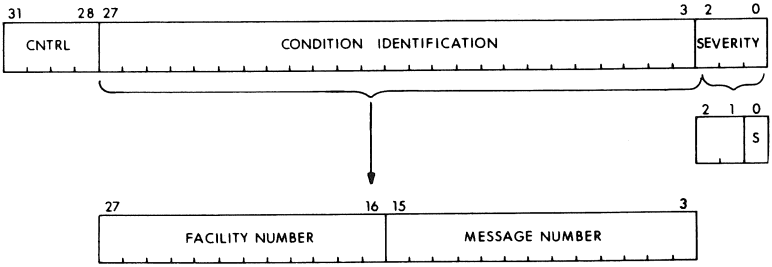


Figure C-2 Condition Value Format

condition identification

Identifies the conditions uniquely on a system-wide basis.

facility

Identifies the software component generating the condition value. Bit 27 is set for customer facilities and clear for DIGITAL facilities.

message number

A status identification, that is, a description of the hardware exception that occurred or a software-defined value. Message numbers with bit 15 set are specific to a single facility. Message numbers with bit 15 clear are system-wide status codes.

severity

The severity code bit 0 is set for success (logical true) and clear for failure (logical false), bits 1 and 2 distinguish degrees of success or failure. The three bits, 0 through 2, taken as an unsigned integer, are interpreted as follows:

severity

The severity code bit 0 is set for success (logical true) and clear for failure (logical false), bits 1 and 2 distinguish degrees of success or failure. The three bits, 0 through 2, taken as an unsigned integer, are interpreted as follows:

STS$K_WARNING	0 = warning
STS$K_SUCCESS	1 = success
STS$K_ERROR	2 = error
STS$K_INFO	3 = information
STS$K_SEVERE	4 = severe_error
	5, 6, 7 reserved to DEC

Section 4.1 describes the severity code more fully.

cntrl

Four control bits. Bit 28 inhibits the message associated with the condition value from being printed by the $EXIT system service. This bit is set by the system default handler after it has output an error message using the $PUTMSG system service. It should also be set in the condition value returned by a procedure as a function value, if the procedure has also signaled the condition (so that the condition has been either printed or suppressed). Bits 29 through 31 must be zero; they are reserved for future use by DIGITAL.

Software symbols are defined for these fields as follows:

Mnemonic	Value	Meaning	Field
STS$V_COND_ID	3	position of 27:3	condition identification
STS$S_COND_ID	25	size of 27:3	
STS$M_COND_ID	mask	mask for 27:3	
STS$V_INHIB_MSG	1@28	position for 28	inhibit message on image exit
STS$S_INHIB_MSG	1	size for 28	
STS$M_INHIB_MSG	mask	mask for 28	
STS$V_FAC_NO	16	position of 27:16	facility number
STS$S_FAC_NO	12	size of 27:16	
STS$M_FAC_NO	mask	mask for 27:16	
STS$V_CUST_DEF	27	position for 27	customer facility
STS$S_CUST_DEF	1	size for 27	
STS$M_CUST_DEF	1@27	mask for 27	
STS$V_MSG_NO	3	position of 15:3	message number
STS$S_MSG_NO	13	size of 15:3	
STS$M_MSG_NO	mastk	mask for 15:3	
STS$V_FAC_SP	15	position of 15	facility specific
STS$S_FAC_SP	1	size for 15	
STS$M_FAC_SP	1@15	mask for 15	
STS$V_CODE	3	position of 14:3	message code
STS$S_CODE	12	size of 14:3	
STS$M_CODE	mask	mask for 14:3	
STS$V_SEVERITY	0	position of 2:0	severity
STS$S_SEVERITY	3	size of 2:0	
STS$M_SEVERITY	7	mask for 2:0	
STS$V_SUCCESS	0	position of 0	success
STS$S_SUCCESS	1	size of 0	
STS$M_SUCCESS	1	mask for 0	

4.1 Interpretation of Severity Codes

A severity code of 0 indicates a warning. This code is used whenever a procedure produces output, but the output might not be what the user expected, for example, a compiler modification of a source program.

A severity code of 1 indicates that the procedure generating the condition value completed successfully, that is, as expected.

A severity code of 2 indicates that an error has occurred, but that the procedure did produce output. Execution can continue but the results produced by the component generating the condition value are not all correct.

A severity code of 3 indicates that the procedure generating the condition value successfully completed, but has some parenthetical information to be included in a message if the condition is signaled.

A severity code of 4 indicates that a severe_error occurred and the component generating the condition value was unable to produce output.

When designing a procedure the choice of severity code for its condition values should be based on the following default interpretations. The calling program typically performs a low bit test, so it treats warnings, errors, and severe_errors as failures, and success and information as successes. If the condition value is signaled (see Section 10.3), the default handler treats severe_errors as reason to terminate and all the others as the basis for attempting to continue. When the program image exits, the command interpreter by default treats errors and severe_errors as the basis for stopping the job, and warnings, information, and successes as the basis for continuing.

The following table summarizes the default interpretation of condition values:

Severity	Routine	Signal	Default at Program Exit
success	normal	continue	continue
information	normal	continue	continue
warning	failure	continue	continue
error	failure	continue	stop job
severe_error	failure	exit	stop job

The default for signaled messages is to output a message to file SYS$OUTPUT. In addition, for severities other than success (STS$K_SUCCESS) a copy of the message is made on file SYS$ERROR. At program exit, success and information completion values do not generate messages, while warning, error, and severe_error condition values generate messages to both files SYS$OUTPUT and SYS$ERROR, unless bit 28 (STS$V_INHIB_MSG) is set.

Unless there is good basis for another choice, a procedure should use either success or severe_error as its severity for each condition value.

4.2 Use of Condition Values

VAX software components return condition values when they complete execution. When a severity code of warning, error, or severe_error is generated, the status code describes the nature of the problem. This value can be tested to change the flow of control of a procedure and/or be used to generate a message. User procedures can also generate condition values to be examined by other procedures and by the command interpreter. User-generated values should set bit 27 and bit 15 so these condition values will not conflict with values generated by DIGITAL.

5. REGISTER USAGE

The following registers have defined uses:

Register	**Use**
PC	Program counter.
SP	Stack pointer.
FP	Current stack frame pointer. It must always point at the current frame. No modification is permitted within a procedure body.
AP	Argument pointer. When a call occurs, AP must point to a valid argument list. A procedure without parameters points to an argument list consisting of a single longword containing the value 0.
R1	Environment value. When a procedure that needs an environment value is called, the calling program must set R1 to the environment value. See bound procedure value in Section 7.3.
R0, R1	Function value return registers. These registers are not to be preserved by any called procedure. They are available to any called procedure as temporary registers.

Registers R2 through R11 are to be preserved across procedure calls. The called procedure can use registers R2 through R11 provided it saves and restores them using the procedure entry mask mechanism. The entry mask mechanism must be used so that any stack unwinding done by the condition handling mechanism will correctly restore all registers. In addition, PC, SP, FP, and AP are always preserved by the CALL instructions and restored by the RET instruction. However, AP can be used as a temporary register by a called procedure.

6. STACK USAGE

The stack frame created by the CALLG/CALLS instructions for the called procedure is:

```
condition handler (0)      :(SP):(FP)
mask/PSW
AP
FP
PC
R2           (optional)
   .
   .
   .
R11          (optional)
```

FP always points at the condition handler longword of the stack frame, (see Section 9). Other use of FP within a procedure is prohibited.

The contents of the stack located at addresses higher than the mask/PSW longword belong to the calling program; they should not be read or written by the called procedure, except as specified in the argument list. The contents of the stack located at addresses lower than SP belong to interrupt and execution routines; they are continually and unpredictably modified.

The called procedure allocates local storage by subtracting the required number of bytes from the SP provided on entry. This local storage is automatically freed by the RET instruction.

Bit 28 of the mask/PSW longword is reserved to DIGITAL for future extensions to the stack frame.

7. ARGUMENT DATA TYPES

Each data type implemented for a higher-level language uses one of the following VAX data types for procedure parameters and elements of file records. When existing data types are not sufficient to satisfy the semantics of a language, new data types will be added to this standard, including certain language-specific ones.

This section also indicates the spelling and punctuation that is used for the name of each data type. In running text, the data type names are not capitalized, except as shown. Also, they are not normally indicated in bold face, italics, or underlined.

Data types fall into three categories: atomic, string, and miscellaneous. These data types can generally be passed by immediate value (if 32 bits or less), by reference or by descriptor. The encoding given in this section is used whenever it is necessary to identify data types, such as in a descriptor. Unless explicitly stated otherwise, all data types represent signed quantities.

NOTE

The unsigned quantities throughout this standard do not allocate space for the sign. All bit or character positions are used for significant data.

7.1 Atomic Data Types

Atomic data types are defined and are encoded as follows:

DSC$K_DTYPE_Z	0	**unspecified** The calling program has specified no data type. The called procedure should assume the argument is of the correct type.
DSC$K_DTYPE_BU	2	**byte logical** 8-bit unsigned quantity.
DSC$K_DTYPE_WU	3	**word logical** 16-bit unsigned quantity.
DSC$K_DTYPE_LU	4	**longword logical** 32-bit unsigned quantity.
DSC$K_DTYPE_QU	5	**quadword logical** 64-bit unsigned quantity.
DSC$K_DTYPE_OU	25	**octaword logical** 128-bit unsigned quantity.
DSC$K_DTYPE_B	6	**byte integer** 8-bit signed 2's complement integer.
DSC$K_DTYPE_W	7	**word integer** 16-bit signed 2's complement integer.
DSC$K_DTYPE_L	8	**longword integer** 32-bit signed 2's complement integer.
DSC$K_DTYPE_Q	9	**quadword integer** 64-bit signed 2's complement integer.
DSC$K_DTYPE_O	26	**octaword integer** 128-bit signed 2's complement integer.

DSC$K_DTYPE_F	10	**F_floating** 32-bit F_floating quantity representing a single-precision number.
DSC$K_DTYPE_D	11	**D_floating** 64-bit D_floating quantity representing a double-precision number.
DSC$K_DTYPE_G	27	**G_floating** 64-bit G_floating quantity representing a double-precision number.
DSC$K_DTYPE_H	28	**H_floating** 128-bit H_floating quantity representing a quadruple-precision number.
DSC$K_DTYPE_FC	12	**F_floating complex** Ordered pair of F_floating quantities, representing a single-precision complex number. The lower addressed quantity is the real part, the higher addressed quantity is the imaginary part.
DSC$K_DTYPE_DC	13	**D_complex** Ordered pair of D_floating quantities, representing a double-precision complex number. The lower addressed quantity is the real part, the higher addressed quantity is the imaginary part.
DSC$K_DTYPE_GC	29	**G_floating complex** Ordered pair of G_floating quantities, representing a double-precision complex number. The lower addressed quantity is the real part, the higher addressed quantity is the imaginary part.

DSC$K_DTYPE_HC	30	**H_floating complex** Ordered pair of H_floating quantities, representing a quadruple-precision complex number. The lower addressed quantity is the real part, the higher addressed quantity is the imaginary part.
DSC$K_DTYPE_CIT	31	**COBOL Intermediate Temporary** A floating-point datum with an 18-digit normalized decimal fraction and a 2-decimal-digit exponent. The fraction is a packed decimal string. The exponent is a 16-bit 2's complement integer (see Section 7.4 for more detail).

7.2 String Data Types

String data types are ordinarily described by a string descriptor. The string data types are defined and are encoded as follows:

Symbol	Code	Name/Description
DSC$K_DTYPE_T	14	**character-coded text** A single 8-bit chacter (atomic data type) or a sequence of 0 to $2^{16}-1$ 8-bit characters (string data type).
DSC$K_DTYPE_NU	15	**numeric string, unsigned**
DSC$K_DTYPE_NL	16	**numeric string, left separate sign**
DSC$K_DTYPE_NLO	17	**numeric string, left overpunched sign**
DSC$K_DTYPE_NR	18	**numeric string, right separate sign**
DSC$K_DTYPE_NRO	19	**numeric string, right overpunched sign**
DSC$K_DTYPE_NZ	20	**numeric string, zoned sign**
DSC$K_DTYPE_P	21	**packed decimal string**

Symbol	Code	Name/Description
DSC$K_DTYPE_V	1	**bit** An aligned bit string. A string of 0 to $2^{16}-1$ contiguous bits. The first bit is bit 0 of the first byte and the last bit is any bit in the last byte. Remaining bits in the last byte must be zero on read and are cleared on write. Unlike the bit unaligned (VU) data type, when the bit (V) data type is used in array descriptors the ARSIZE field is in units of bytes, not bits, since allocation is a multiple of 8 bits.
DSC$K_DTYPE_VU	34	**bit unaligned** The data are 0 to $2^{16}-1$ contiguous bits located arbitrarily with respect to byte boundaries. See also bit (V) data type. Because additional information is required to specify the bit position of the first bit, this data type can only be used with the unaligned bit string and unaligned bit array descriptors (see Section 8.-14 and 8.15).

7.3 Miscellaneous Data Types

Miscellaneous data types are defined and are encoded as follows:

Symbol	Code	Name/Description
DSC$K_DTYPE_ZI	22	**sequence of instructions**
DSC$K_DTYPE_ZEM	23	**procedure entry mask**
DSC$K_DTYPE_DSC	24	**descriptor** This data type allows a descriptor to be a data type; thus, levels of descriptors are allowed.

DSC$K_DTYPE_BPV	32	**bound procedure value** A two-longword entity in which the first longword contains the address of a procedure entry mask and the second longword is the environment value. The environment value is determined in a language-specific manner when the original bound procedure value is generated. When the bound procedure is called, the calling program loads the second longword into R1. When the environment value is not needed, this data type can be passed using the immediate value mechanism. In this case, the argument list entry contains the address of the procedure entry mask and the second longword is omitted.
DSC$K_DTYPE_BLV	33	**bound label value** A two-longword entity in which the first longword contains the address of an instruction and the second longword is the language-specific environment value. The environment value is determined in a language-specific manner when the original bound label value is generated.

The type codes 35 through 191 are reserved to DIGITAL. Codes 192 through 255 are reserved for DIGITAL's Computer Special Systems Group and for customers for their own use.

7.4 COBOL Intermediate Temporary Data Type

A COBOL intermediate temporary datum is 12 contiguous bytes starting on an arbitrary byte boundary. It is specified by its address A.

15 12	11 8	7 4	3 0	
EXPONENT				: A
f<16>	f<15>	0	f<17>	: A+2
f<12>	f<11>	f<14>	f<13>	: A+4
f<8>	f<7>	f<10>	f<9>	: A+6
f<4>	f<3>	f<6>	f<5>	: A+8
f<0>	SIGN	f<2>	f<1>	: A+10

Figure C-3 COBOL Intermediate Temporary Datum

A COBOL intermediate temporary datum represents a floating point datum with a normalized 18-digit packed decimal fraction and a 16-bit 2's complement integer exponent. Bytes 0 and 1 are the exponent. Bytes 2 through 11 contain the normalized packed decimal fraction. The sign of the datum is the sign of the fraction. If the fraction is zero, the value of the datum is zero.

If the exponent is from −99 to +99, operations can be performed on this datum. If the exponent is outside this range, a reserved operand condition is signaled (see Section 10). If a calculated datum has an exponent greater than +99, the exact result with the low-order 15 bits of the true exponent is stored in the result datum and an overflow condition is signaled.

If a calculated datum has an exponent less than −99, the exact result with the low-order 15 bits of the true exponent is stored in the result datum and an underflow condition is signaled. The condition handler can take the appropriate action. Condition mnemonics have a COB$_ prefix and are documented with the COBOL part of the Run-Time Library. An exponent value of −32,768 is taken as reserved and should be used to encode reserved operands such as uninitialized datum, indeterminate value, etc. By convention, if the fraction of a result is 0, the exponent is set to 0. Fractions are generated with preferred sign codes and avoid −0.

8. ARGUMENT DESCRIPTOR FORMATS

A uniform descriptor mechanism is defined for use by all procedures that conform to the VAX Procedure Calling Standard. Descriptors are self-describing and the mechanism is extensible. When existing descriptors are not sufficient to satisfy the semantics of a language, new descriptors will be added to this standard.

Unless explicitly stated otherwise in this standard, the calling program fills in all fields in descriptors. This is true whether the descriptor is generated by default or by a language extension. Fields are filled in even if a called procedure written in the same language would ignore the contents of some of the fields. A descriptor conforms to this standard if all fields are filled in by the calling program according to the standard, even if the field is not needed by the called program.

NOTE

> Unless explicitly stated otherwise, all fields in descriptors represent unsigned quantities, are read-only from the point of view of the called procedure, and may be allocated in read-only memory at the option of the calling program.

If a language processor implements a language-specific data type that is not added to this standard (see Section 7), it is not required to use a standard descriptor to pass an array of this data type. However, if it does pass an array of such a data type using a standard descriptor, it will fill in the DSC$B_DTYPE field with 0 indicating that the data type field is unspecified, rather than using a more general data type code. For example, an array of P.L/I POINTER data types has the DTYPE field filled in with 0 rather than 4 (longword logical). The remaining fields are filled in as specified by this standard, that is, DSC$W_LENGTH is filled in with the size in bytes, etc. Since it is conceivable that the language-specific data type might be added to the standard in the future, generic application procedures that examine the DTYPE field should be prepared for 0 and for additional data types to be added in the future.

8.1 Descriptor Prototype

Each class of descriptors consists of at least two longwords in the following format:

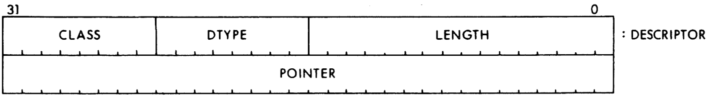


Figure C-4 Decriptor Prototype

Symbol	Description
DSC$W_LENGTH <0,15:0>	A one-word field specific to the descriptor class, typically a 16-bit (unsigned) length.

DSC$B_DTYPE <0,23:16>	A one-byte data type code (see Section 7).
DSC$B_CLASS <0,31:24>	A one-byte descriptor class code (see 8.2 through 8.11).
DSC$A_POINTER <1,31:0>	A longword containing the address of the first byte of the data element described.

Note that the descriptor can be placed in a pair of registers with a MOVQ instruction and then the length and address can be used directly. This gives a word length, so the class and type are placed as bytes in the rest of that longword. When the class field is zero, no more than the above information can be assumed.

8.2 Scalar, String Descriptor (DSC$K_CLASS_S)

A single descriptor form is used for scalar data and fixed-length strings. Any VAX data type can be used with this descriptor, except data type 34 (bit unaligned).

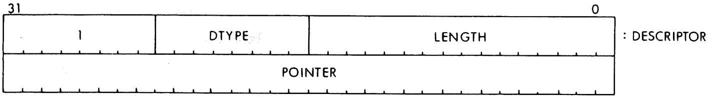


Figure C-5 Scalar, String Descriptor

Symbol	**Description**
DSC$W_LENGTH	Length of data in bytes, unless the DSC$B_DTYPE field contains the value 1 (bit) or 21 (packed decimal). Length of data item is in bits for bit. Length of data item is the number of 4-bit digits (not including the sign) for packed decimal string.
DSC$B_DTYPE	A one-byte data type code (see Section 7).
DSC$B_CLASS	1 = DSC$K_CLASS_S.
DSC$A_POINTER	Address of first byte of data storage.

If the data type is 14 (character-coded text) and the string must be extended in a string comparison or is being copied to a fixed length string containing a greater length, the space character (20_{16} if ASCII) is used as the fill character.

8.3 Dynamic String Descriptor (DSC$K_CLASS_D)

A single descriptor form is used for dynamically allocated strings. When a string is written, either or both the length field and the pointer field can be changed. The VAX Run-Time Library provides procedures for changing fields. As an input parameter this format is interchangeable with class 1 (DSC$K_CLASS_S).

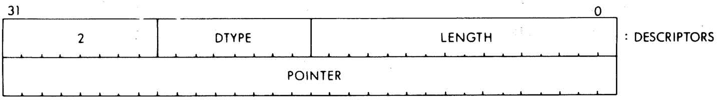


Figure C-6 Dynamic String Descriptor

DSC$W_LENGTH — Length of data item in bytes, unless the DSC$B_DTYPE field contains the value 1 (bit) or 21 (packed decimal). Length of data item is in bits for bit. Length of data item is the number of 4-bit digits (not including the sign) for packed decimal string.

DSC$B_DTYPE — A one-byte data type code (see Section 7).

DSC$B_CLASS — 2 = DSC$K_CLASS_D.

DSC$A_POINTER — Address of first byte of data storage.

8.4 Variable Buffer Descriptor (DSC$K_CLASS_V)

Reserved for use by DIGITAL.

8.5 Array Descriptor (DSC$K_CLASS_A)

The array descriptor is used to describe contiguous arrays of atomic data type or contiguous arrays of fixed length strings. An array descriptor consists of three contiguous blocks. The first block contains the descriptor prototype information and is part of every array descriptor. The second and third blocks are optional. If the third block is present, so is the second. A complete array descriptor has the form:

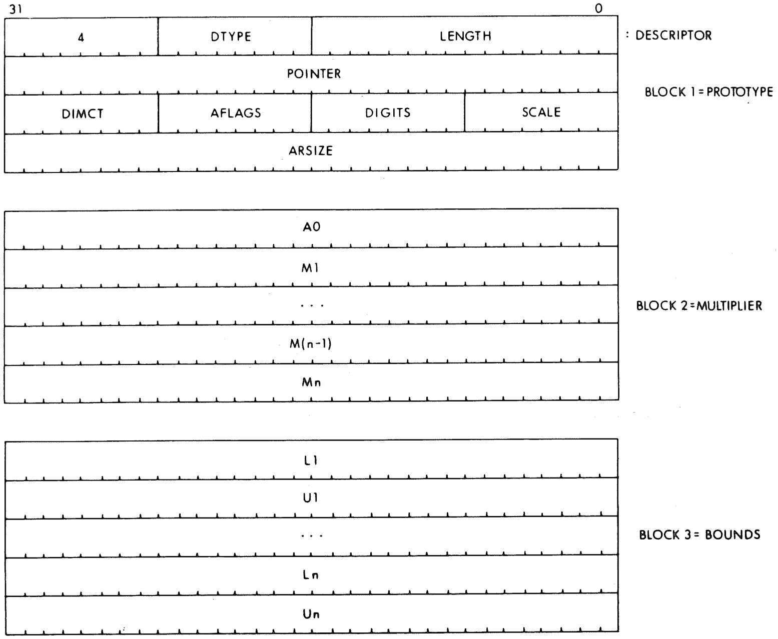


Figure C-7 Array Descriptor

Symbol	Description
DSC$W_LENGTH	Length of an array element in bytes, unless the DSC$B_DTYPE field contains the value 1 (bit) or 21 (packed decimal). Length of an array element is in bits for bit. Length of an array element is the number of 4-bit digits (not including the sign) for packed decimal string.
DSC$B_DTYPE	A one-byte data type code (see Section 7).
DSC$B_CLASS	4 = DSC$K_CLASS_A.
DSC$A_POINTER	Address of first actual byte of data storage.

Symbol	Description
DSC$B_SCALE <2,7:0>	Signed power of ten multiplier to convert the internal form to external form. For example, if internal number is 123 and scale is +1, then the external number is 1230.
DSC$B_DIGITS <2,15:8>	If non-zero, unsigned number of decimal digits in the internal representation. If zero, the number of digits can be computed based on DSC$W_LENGTH.
DSC$B_AFLAGS <2,23:16>	Array flag bits:
Reserved <2,19:16>	Must be zero.
DSC$V_FL_REDIM <2,20>	If set, the array can be redimensioned, that is DSCA_AO, DSCL_Mi, DSC$L_Li, and DSC$L_Ui may be changed. The redimensioned array cannot exceed the size allocated to the array (DSC$L_ARSIZE).
DSC$V_FL_COLUMN <2,21>	If set, the elements of the array are stored by columns (FORTRAN). That is, the leftmost subscript (first dimension) is varied most rapidly, and the rightmost subscript (nth dimension) is varied least rapidly. If not set, the elements are stored by rows (most other languages). That is, the rightmost subscript is varied most rapidly and the leftmost subscript is varied least rapidly.
DSC$V_FL_COEFF <2,22>	If set, the multiplicative coefficients in Block 2 are present. Must be set if DSC$V_FL_BOUNDS is set.

DSC$V_FL_BOUNDS <2,23>
If set, the bounds information in Block 3 is present and requires that DSC$V_FL_COEFF be set.

DSC$B_DIMCT <2,31:24>
Number of dimensions, n.

DSC$L_ARIZE <3,31:0>
Total size of array (in bytes unless the DSC$B_TYPE field contains the value 21, see description for DSC$W_LENGTH). A redimensioned array may use less than the total size allocated.

For data type 1 (bit), DSC$W_LENGTH is in bits while DSC$L_ARIZE is in bytes since the unit of length in bits while the unit of allocation is aligned bytes.

DSC$A_A0 <4,31:0>
Address of element A(0,0,...,0). This need not be within the actual array. It is the same as DSC$A_POINTER for zero-origin arrays.

DSC$L_Mi <4+i,31:0>
Addressing coefficients. (Mi = Ui−Li+1)

DSC$L_Li <3+n+2*i,31:0>
Lower bound (signed) of ith dimension.

DSC$L_Ui <4+n+2*i,31:0>
Upper bound (signed) of ith dimension.

The following formulas specify the effective address, E, of an array element. WARNING: modification of the following formulas is required if DSC$B_DTYPE contains a 1 or 21 because DSC$W_LENGTH is given in bits or 4-bit digits rather than bytes.

The effective address, E, For element A(I):

$$E = A0 + I*LENGTH$$
$$= POINTER + [I - L_1]*LENGTH$$

The effective address, E, for element A(I_1, I_2) with DSC$V_FL_COLUMN clear:

$$\begin{aligned} E &= A0 + [I_1{*}M_2 + I_2]{*}LENGTH \\ &= POINTER + [[I1-Li]{*}M2 + I2 - L2]{*}LENGTH \end{aligned}$$

The effective address, E, for element $A(I_1, I_2)$ with DSC$V_FL_COLUMN set:

$$\begin{aligned} E &= A0 + [I_2{*}M_1 + I_1]{*}LENGTH \\ &= POINTER + [[I_2-L_2]{*}M_1 + I_1 - L_1]{*}LENGTH \end{aligned}$$

The effective address E, for element $A(I_1, \ldots, I_n)$ with DSC$V_FL_COLUMN clear:

$$\begin{aligned} E &= A0 + [[[[\ldots[I_1{*}M_2 + \ldots]{*}M_{n-2} + I_{n-2}]{*}M_{n-1} + I_{n-1}]{*}M_n \\ &\quad + I_n]{*}LENGTH \\ &= POINTER + [[[[\ldots[I_1 - L_1]{*}M_2 + \ldots]{*}M_{n-2} + I_{n-2} \\ &\quad - L_{n-2}]{*}M_{n-1} + I_{n-1} - L_{n-1}]{*}M_n + I_n - L_n]{*}LENGTH \end{aligned}$$

The effective address, E, for element $A(I_1, \ldots, I_n)$ with DSC$V_FL_COLUMN set:

$$\begin{aligned} E &= A0 + [[[[\ldots[I_n]{*}M_{n-1} + \ldots]{*}M_3 + I_3]{*}M_2 + I_2]{*}M_1 + I_1] \\ &\quad {*}LENGTH \\ &= POINTER + [[[[\ldots[I_n - L_n]{*}M_{n-1} + \ldots]{*}M_3 + I_3 - L_3]{*}M_2 \\ &\quad + I_2 - L_2]{*}M_1 + I_1 - L_1]{*}LENGTH \end{aligned}$$

8.6 Procedure Descriptor (DSC$K_CLASS_P)

The descriptor for a procedure specifies its entry address and function value data type, if any. A procedure descriptor has the form:

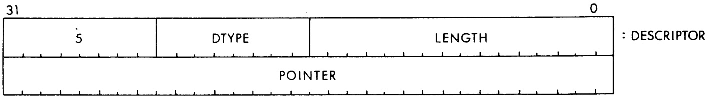


Figure C-8 Procedure Descriptor

Symbol	Description
DSC$W_LENGTH	Length associated with the function value or 0 if no function value is returned.
DSC$B_DTYPE	Function value data type code (see Section 7).
DSC$B_CLASS	5 = DSCK$K_CLASS_P.
DSC$A_POINTER	Address of entry mask to routine.

Procedures return a function value in R0, R1/R0, or using the first argument list entry depending on the size of the data type (see Section 3).

8.7 Procedure Incarnation Descriptor (DSC$K_CLASS_PI)

Obsolete.

8.8 Label Descriptor (DSC$L_CLASS_J)

Reserved for use by the VAX Debugger.

8.9 Label Incarnation Descriptor (DSC$K_CLASS_JI)

Obsolete.

8.10 Decimal Scalar String Descriptor (DSC$K_CLASS_SD)

Decimal size and scaling information for scalar data and simple strings is given in a single descriptor form as follows:

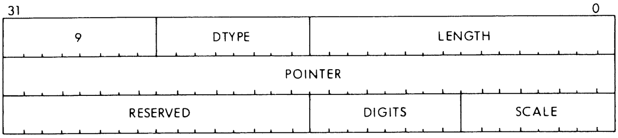


Figure C-9 Decimal Scalar String Descriptor

Symbol	Description
DSC$W_LENGTH	Length of data item in bytes, unless the DSC$B_DTYPE field contains the value 1 (bit) or 21 (packed decimal). Length of data item is in bits for bit. Length of data item is the number of 4-bit digits (not including the sign) for packed decimal string.
DSC$B_DTYPE	A one-byte data type code (see Section 7).
DSC$B_CLASS	9 = DSC$K_CLASS_SD.
DSC$A_POINTER	Address of first byte of data storage.
DSC$B_SCALE	Signed power of ten multiplier to convert the internal form to external form. For example, if

internal number is 123 and scale is +1, then the external number is 1230.

DSCC$B_DIGITS

If nonzero, unsigned number of decimal digits in the internal representation. If zero, the number of digits can be computed based on DSC$W_LENGTH.

Reserved <2,31:16>

Reserved for future use. Must be zero.

8.11 Noncontiguous Array Descriptor (DSC$K_CLASS_NCA)

The noncontiguous array descriptor describes an array where the storage of the array elements may be allocated with a fixed, nonzero number of bytes separating logically adjacent elements. Two elements are said to be logically adjacent if their subscripts differ by one in the most rapidly varying dimension only. The difference between the addresses of two adjacent elements is termed the stride. Whether elements are stored by row or column is the option of the calling program and is automatically taken care of by a single accessing algorithm used by the called procedure.

This array descriptor is to be used where the calling program, at its option, can pass a slice of an array which contains noncontiguous allocation. At the present time this standard indicates no preference between the noncontiguous array descriptor (NCA) and the contiguous array descriptor (A) as described in Section 8.5 for language processors that always allocate arrays that are contiguous.

Symbol

Description

DSC$W_LENGTH

Length of an array element in bytes, unless the DSC$B_DTYPE field contains the value 1 (bit) or 21 (packed decimal). Length of an array element is in bits for bit. Length of an array element is the number of 4-bit digits (not including the sign) for packed decimal string.

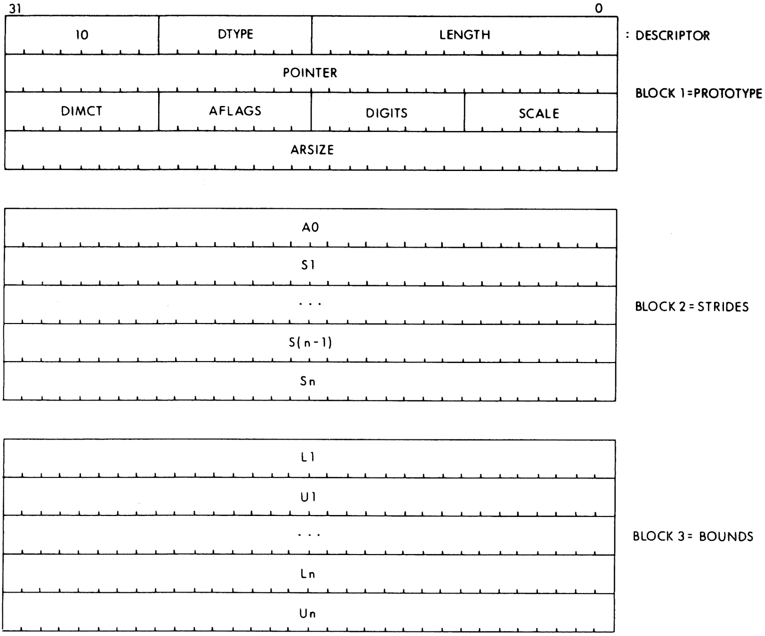


Figure C-10 Noncontiguous Array Descriptor

Symbol	Description
DSC$B_DTYPE	A one-byte data type code (see Section 7).
DSC$B_CLASS	10 = DSC$K_CLASS_NCA.
DSC$A_POINTER	Address of first actual byte of data storage.
DSC$B_SCALE <2,7:0>	Signed power of ten multiplier to convert the internal form to the external form. For example, if the internal number is 123 and scale is +1, then the external number is 1230.

Symbol — **Description**

DSC$B_DIGITS <2,15:8>
If nonzero, unsigned number of decimal digits in the internal representation. If zero, the number of digits can be computed based on DSC$W_LENGTH.

DSC$B_AFLAGS <2,23:16>
Array flag bits:

- Reserved <2,19:16>
  Must be zero. Reserved for future standardization by DIGITAL.

- DSC$_FL_REDIM <2,20>
  Must be zero.

- Reserved <2,23:21>
  Must be zero. Reserved for future standardization by DIGITAL.

DSC$B_DIMCT <2,31:24>
Number of dimensions, n.

DSC$L_ARSIZE <3,31:0>
If the elements are actually contiguous then ARSIZE is the total size of the array (in bytes, unless the DSC$B_DTYPE field contains the value 21, see description of DSC$W_LENGTH). If the elements are not allocated contiguously or the program unit allocating the descriptor is uncertain whether the array is actually contiguous or not, the value placed in ARSIZE may be meaningless.

For data type 1 (bit), DSC$W_LENGTH is in bits while DSC$L_ARSIZE is in bytes since the unit of length is in bits while the unit of allocation is aligned bytes.

DSC$A_A0 <4,31:0>
Address of element A(0,0,...,0). This need not be within the

Symbol	Description
	actual array. It is the same as DSC$A_POINTER for zero-orgin arrays. DSC$A_A0 = POINTER – (S1* L1 + S2*L2 + ... +Sn*Ln)
DSC$L_Si <4+i,31:0>	Stride of the ith dimension. The difference between the addresses of successive elements of the ith dimension.
DSC$L_Li <3+n+2*i,31:0>	Lower bound (signed) of the ith dimension.
DSC$L_Ui <4+n+2*i,31:0>	Upper bound (signed) of the ith dimension.

The following formulas specify the effective address, E, of an array element. WARNING: modification of the following formulas is required if DSC$B_DTYPE equals 1 or 21 because DSC$W_LENGTH is given in bits or 4-bit digits rather than bytes.

The effective address, E, of A(I):

$$\begin{aligned} E &= A0 + S_1{*}I \\ &= POINTER + S_1{*}[I - L_1] \end{aligned}$$

The effective address, E, of A(I_1,I_2):

$$\begin{aligned} E &= A0 + S_1{*}I_1 + S_2{*}I_2 \\ &= POINTER + S_1{*}[I_1 - L_1] + S_2{*}[I_2 - L_2] \end{aligned}$$

The effective address, E, of A(I_1, . . . ,I_n):

$$\begin{aligned} E &= A0 + S_1{*}I_1 + \ldots + S_n{*}I_n \\ &= POINTER + S_1{*}[I_1 - L_1] + \ldots + S_n{*}[I_n - L_n] \end{aligned}$$

8.12 Varying String Descriptor (DSC$K_CLASS_VS)

A single descriptor form is used for varying strings consisting of two fixed-length areas allocated contiguously with no padding between:

1. CURLEN - An unsigned word specifying the current length in bytes of the immediately following string (byte aligned).
2. BODY - A fixed length area containing the string which can vary from 0 to a maximum length defined for each instance of string.

As an input parameter, this format is not interchangeable with class 1 (DSC$K_CLASS_S) and 2 (DSC$K_CLASS_D). When a called procedure modifies a varying string passed by reference or by descriptor, it writes the new length, n, into CURLEN and may modify all bytes of BODY.

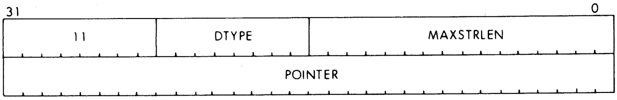


Figure C-11 Varying String Descriptor

Symbol	**Description**
DSC$W_MAXSTRLEN	Max length of the BODY field of the varying string in bytes in the range 0 to 2**16-1.
DSC$B_DTYPE	A one-byte data type code that must be 14 specifying the data type of the BODY (indicating character-coded text - see section 7.2). The use of other data types is reserved for future standardization.
DSC$B_CLASS	11 = DSC$K_CLASS_VS.
DSC$A_POINTER	Address of first field (CURLEN) of the varying string.

Example: MAXSTRLEN contains 7, CURLEN contains 3, string is currently "ABC", and the remaining 4 bytes are currently undefined:

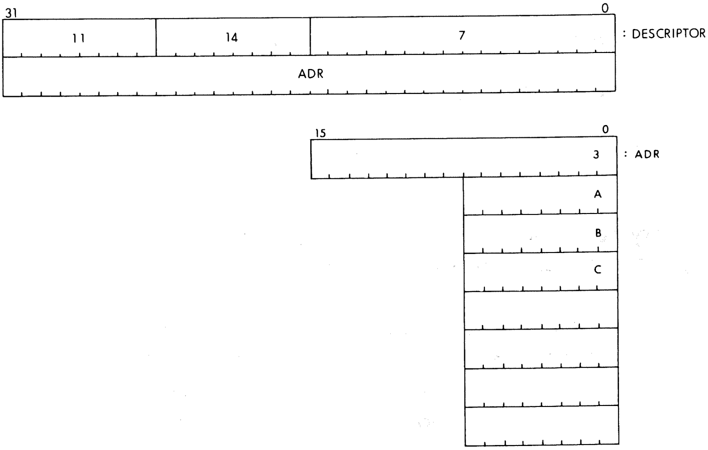


Figure C-12 Varying String Example

8.13 Varying String Array Descriptor (DSC$K_CLASS_VSA)

A variant of the noncontiguous array descriptor is used to specify an array of varying strings where each varying string has the same maximum length. Each array element is a varying string, that is two fixed-length areas allocated contiguously with no padding between:

1. CURLEN - An unsigned word specifying the current length in bytes of the immediately following string (byte aligned).
2. BODY - A fixed length area containing the string which can vary from 0 to the maximum length defined for an array element (MAXSTRLEN).

When a called procedure modifies a varying string in an array of varying strings passed to it by reference or by descriptor, it writes the new length, n, into CURLEN and may modify all bytes of BODY. The format of this descriptor is the same as the noncontiguous array descriptor except:

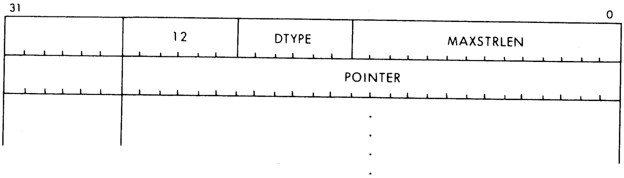


Figure C-13 Varying String Array Descriptor

Symbol	Description
DSC$W_MAXSTRLEN	Max. length of the BODY field of an array element in bytes in the range 0 to 2**16-1.
DSC$B_DTYPE	A one-byte string data type code that must be 14 specifying the data type of the BODY fields of the varying array (indicating character coded-text - see Section 7.2). The use of other data types is reserved for future standardization.
DSC$B_CLASS	12 = DSC$K_CLASS_VSA.
DSC$A_POINTER	Address of first actual byte of data storage.

The remaining fields are identical to the noncontiguous array descriptor. The effective address computation of an array element produces the address of CURLEN of the desired element.

8.14 Unaligned Bit String Descriptor (DSC$K_CLASS_UBS)

A descriptor is used to pass an unaligned bit string (DSC$K_DTYPE_VU) that starts on an arbitrary bit boundary and ends on an arbitrary bit boundary.

The length is 0 to 2**16-1 bits. The bit string may be accessed directly using the VAX variable bit field instructions. Therefore, the descriptor provides two components: a base address and a signed relative bit position.

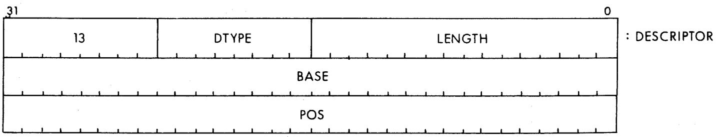


Figure C-14 Unaligned Bit String Descriptor

Symbol	Description
DSC$W_LENGTH	Length of data item in bits.
DSC$B_DTYPE	A one-byte data type code that must be 34 indicating bit unaligned data type (see Section 7). The use of other data types is reserved for future standardization.
DSC$B_CLASS	13 = DSC$K_CLASS=UBS
DSC$A_BASE	Base of address relative to which the signed relative bit position, POS, is used to locate the bit string. The base address need not be first actual byte of data storage.
DSC$L_POS	Signed longword relative bit position with respect to BASE of the first bit of unaligned bit string.

Example: A called procedure can use the following instruction to access a bit string of 32 bits or less. If R0 contains the address of the descriptor, the following instruction copies the bit string to R1:

Example: A called procedure can use the following instruction to access a bit string of 32 bits or less. If R0 contains the address of the descriptor, the following instruction copies the bit string to R1:

```
EXTZV   DSC$L_POS(R0),                    DSC$W_LENGTH(R0),
@DSC$A_BASE(R0), R1
```

8.15 Unaligned Bit Array Descriptor (DSC$K_CLASS_UBA)

A variant of the noncontiguous array descriptor is used to specify an array of unaligned bit strings. Each array element is a bit unaligned data type (DSC$K_TYPE_VU) that starts on an arbitrary bit boundary and ends on an arbitrary bit boundary. The length of each element is the same and is 0 to $2^{16}-1$ bits. Elements of the array may be accessed directly using the VAX variable bit field instructions. Therefore, the descriptor provides two components: a byte address, DSC$A_BASE, and a means to compute the signed bit offset, EB, with respect to BASE of an array element.

The unaligned bit array descriptor consists of 4 contiguous blocks that are always present. The first block contains the descriptor prototype information. A complete unaligned bit array descriptor has the form.

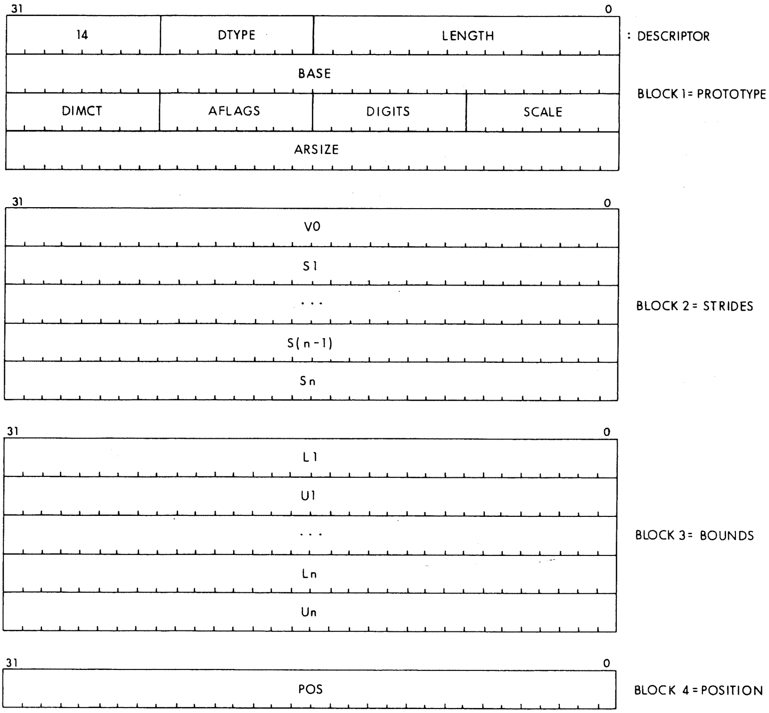


Figure C-15 Unaligned Bit Array Descriptor

Symbol	Description
DSC$W_LENGTH	Length of an array element in bits.

Symbol	Code Name/Description
DSC$B_DTYPE	A one byte data type code that must be 34 indicating bit unaligned data type (see Section 7). The use of other data types is reserved for future standardization.
DSC$B_CLASS	14 = DSC$K_CLASS_UBA
DSC$A_BASE	Base address relative to which the effective bit offset, EB, is used to locate elements of the array. The base address need not be the first actual byte of data storage.
DSC$B_SCALE <2,7:0>	Must be zero. Reserved for future standardization by DIGITAL.
DSC$B_DIGITS <2,15:8>	Must be zero. Reserved for future standardization by DIGITAL.
DSC$B_AFLAGS <2,23:16>	Array flag bits.
Reserved <2,19:16>	Must be zero. Reserved for future standardization by DIGITAL.
DSC$V_FL_REDIM <2,20>	Must be zero.
Reserved <2,23:21>	Must be zero. Reserved for future standardization by DIGITAL.
DSC$B_DIMCT <2,31:24>	Number of dimensions, n.
DSC$L_ARSIZE <3,31:0>	If the elements are actually allocated contiguously, then ARSIZE is the total size of the array in bits. If the elements are not allocated contiguously or the program unit allocating the descriptor is uncertain whether the array is actually contiguous or not, the value placed in ARSIZE may be meaningless.
DSC$L_V0 <4,31:0>	Signed bit offset of element A(0,...,0) with respect to BASE. V0 = POS - [S1*L1 + ... + Sn*Ln].
DSC$L_Si <4+i,31:0>	Stride of the ith dimension. The difference between the bit (not byte) addresses of successive elements of the ith dimension.

Symbol	Description
DSC$L_L1 <3+n+2*i,31:0>	Lower bound (signed) of the ith dimension.
DSC$L_Ui <4+n+2*i,31:0>	Upper bound (signed) of the ith dimension.
DSC$L_POS <5+n*3,31:0>	Signed longword relative bit position with respect to BASE of the first actual bit of the array, that is element A(L1,...,Ln).

The signed, 32-bit effective bit offset, EB, of $A(I_1)$:

$$\begin{aligned} EB &= V0 + S_1{*}I_1 \\ &= POS + S_1{*}[I_1 - L_1] \end{aligned}$$

The signed, 32-bit effective bit offset, EB, of $A(I_1,I_2)$:

$$\begin{aligned} EB &= V0 + S_1{*}I_1 + S_2{*}I_2 \\ &= POS + S_1{*}[I_1 - L_1] + S_2{*}[I_2 - L_2] \end{aligned}$$

The signed, 32-bit effective bit offset, EB, of $A(I_1, \ldots , I_n)$:

$$\begin{aligned} EB &= V0 + S_1{*}I_1 + \ldots + S_n{*}I_n \\ &= POS + S_1{*}[I_1 - L_1] + \ldots + S_n{*}[I_n - L_n] \end{aligned}$$

Note: EB is computed ignoring integer overflow. EB is then usable as the position operand and the contents of DSC$A_BASE is usable as the base address operand in the VAX variable length bit field instructions. Therefore, BASE must specify a byte that is within 2**28 bytes of all bytes of storage in the bit array.

Example: Consider a one-origin, one-dimension, 5-element array consisting of 3-bit elements allocated adjacently (therefore S1 = 3). Assume BASE is byte 1000 and the first actual element, A(1), starts at bit 4 of byte 1001.

The array would look like:

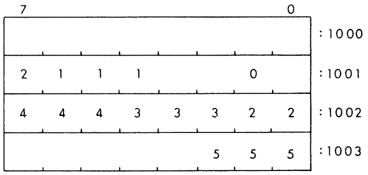


Figure C-16 Array Example

The following dependent field values occur in the descriptor:

POS = 12
V0 = 12 - 3*1 = 9

8.16 Reserved Descriptors

Descriptor classes 15 through 191 are reserved for DIGITAL. Classes 192 through 255 are reserved for DIGITAL's Computer Special Systems group and customers.

9. VAX CONDITIONS

A condition is either:

- A hardware-generated synchronous exception
- A software event that is to be processed in a manner analogous to a hardware exception.

Floating-point overflow exception, memory access violation exception, and reserved operation exception are examples of hardware-generated conditions. An output conversion error, an end-of-file, or the filling of an output buffer are examples of software events that might be treated as conditions.

Depending on the condition and on the program, four types of action can be taken when a condition occurs.

1. **Ignore the condition.** For example, if an underflow occurs in a floating-point operation, continuing from the point of the exception with a zero result may be satisfactory.
2. **Take some special action and then continue from the point at which the condition occurred.** For example, if the end of a buffer is reached while a series of data items are being written, the special action is to start a new buffer.
3. **End the operation and branch from the sequential flow of control.** For example, if the end of an input file is reached, the branch exits from a loop that is processing the input data.
4. **Treat the condition as an unrecoverable error.** For example, when the floating divide by zero exception condition occurs, the program exits, after writing (optionally) an appropriate error message.

When an unusual event or errors occurs in a called procedure, the procedure can return a condition value to the caller indicating what has happened (see Section 4). The caller tests the condition value takes the appropriate action.

When an exception is generated by the hardware, a branch out of the program's flow of control occurs automatically. In this case, and for certain software generated events, it is more convenient to handle the condition as soon as it is detected rather than to program explicit tests.

9.1 Condition Handlers

To handle hardware- or software-detected exceptions, the VAX Condition Handling Facility allows the programmer to specify a condition handler procedure to be called when an exception condition occurs. This same handler procedure may also be used to handle software-detected conditions.

An active procedure can establish a condition handler to be associated with it. The presence of a condition handler is indicated by a nonzero address in the first longword of the procedure's stack frame. When an event occurs that is to be treated using the condition handling facility, the procedure detecting the event signals the event by calling the facility and passing a condition value describing the condition that occurred. This condition value has the format and interpretation as described in Section 4. All hardware exceptions are signaled.

When a condition is signaled, the condition handling facility looks for a condition handler in the current procedure's stack frame. If a handler is found, it is entered. If no handler is associated with the current procedure, the immediately preceding stack frame is examined. Again, if a handler is found it is entered. If a handler is not found, the search of previous stack frames continues until the default condition handler established by the system is reached or the stack runs out.

The default condition handler prints messages indicated by the signal argument list by calling the Put Message (SYS$PUTMSG) system service, followed by an optional symbolic stack traceback. Success conditions with STS$K_SUCCESS result in messages to file SYS$OUTPUT only. All other conditions, including informational messages (STS$K_INFO), produce messages on files SYS$OUTPUT and SYS$ERROR.

For example, if a procedure needs to keep track of the occurrence of the floating-point underflow exception, it can establish a condition handler to examine the condition value passed when the handler is invoked. Then when the floating-point underflow exception occurs, the condition handler will be entered and will log the condition. The handler will return to the instruction immediately following the instruction causing the underflow.

If floating-point operations occur in many procedures of a program, the condition handler can be associated with the program's main procedure. When the condition is signaled, successive stack frames are searched until the stack frame for the main procedure is found, at which time the handler will be entered. If a user program has not associated a condition handler with any of the procedures that are active at the time of the signal, successive stack frames will be searched until the frame for the system program invoking the user program is reached. A default condition handler that prints an error message will then be entered.

9.2 Condition Handler Options

Each procedure activation potentially has a single condition handler associated with it. This condition handler will be entered whenever any condition is signaled within that procedure. (It can also be entered as a result of signals within active procedures called by the procedure.) Each signal includes a condition value (see Section 4), which describes the condition causing the signal. When the condition handler is entered, the condition value should be examined to determine the cause of the signal. After the handler has processed the condition or chosen to ignore it, it can:

- Return to the instruction immediately following the signal. Note that it is not always possible to make such a return.
- Resignal the same or a modified condition value. A new search for a condition handler will begin with the immediately preceding stack frame.
- Signal a different condition.
- Unwind the stack.

10. OPERATIONS INVOLVING CONDITION HANDLERS

The functions provided by the VAX Condition Handling Facility are to:

1. **Establish a condition handler.** A condition handler is associated with the current procedure by placing the handler's address in the current procedure's activation stack frame.
2. **Revert to the caller's handling.** If a condition handler has been established, it can be removed by clearing its address in the current procedure activation's stack frame.
3. **Enable or disable certain arithmetic exceptions.** The following hardware exceptions can be enabled or disabled by software: floating-point underflow, integer overflow, and decimal overflow. No signal occurs when the exception is disabled.
4. **Signal a condition.** Signaling a condition initiates the search for an established condition handler.
5. **Unwind the stack.** Upon exit from a condition handler it is possible to remove one or more frames occurring before the signal from the stack. During the unwinding operation, the stack is

scanned and if a condition handler is associated with a frame, that handler is entered before the frame is removed. Unwinding the stack allows a procedure to perform application specific cleanup operations before exiting.

10.1 Establish a Condition Handler

Each procedure activation has a condition handler potentially associated with it using longword 0 in its stack frame. Initially, longword 0 contains 0, indicating no handler. A handler is established by moving the address of the handler's procedure entry point mask to the establisher's stack frame.

In addition, VAX/VMS provides three statically allocated exception vectors for each access mode of a process. These vectors are available to declare condition handlers that take precedence over any handlers declared at the procedure level. These are used, for example, to allow a debugger to monitor all exceptions and for the system to establish a last chance handler. Since these handlers do not obey the procedure nesting rules, they should not be used by modular code. Instead, the stack based declaration should be used.

The code to establish a condition handler is:

```
        MOVAB handler_entry_point,0(FP)
```

10.2 Revert to the Caller's Handling

Reverting to the caller's handling deletes the condition handler associated with the current procedure activation. This is done by clearing the handler address in the stack frame.

The code to revert to the caller's handling is:

```
        CLRL 0(FP)
```

10.3 Signal a Condition

The signal operation is the method used for indicating the occurrence of an exception condition. To issue a message and be able to continue execution after handling the condition, a program calls the LIB$SIGNAL procedure as follows:

```
        CALL LIB$SIGNAL (condition_value, arg_list...)
```

To issue a message, but not continue execution, a program calls LIB$STOP, as follows:

```
        CALL LIB$STOP (condition_value, arg_list...)
```

In both cases, condition_value indicates the condition that is signaled. However, LIB$STOP sets the severity of the condition_value to be a severe_error. The remaining arguments describe the details of the exception. These are the same arguments used to issue a system message.

Note that unlike most calls, LIB$SIGNAL and LIB$STOP preserve R0 and R1 as well as the other registers. Therefore, a debugger can insert a call to LIB$SIGNAL to display the entire state of the process at the time of the exception. It also allows signals to be coded in MACRO without changing the register usage. This feature of preserving R0 and R1 is useful for debugging checks and gathering statistics. Hardware and system service exceptions behave like calls to LIB$SIGNAL.

The signal procedure examines the two exception vectors and then up to 64K previous stack frames and finally the last-chance exception vector, if necessary. The current and previous stack frames are found by using FP and chaining back through the stack frames using the saved FP in each frame. The exception vectors have three address locations per access mode.

As a part of image start-up, the system declares a default last-chance handler. This handler is used as a last resort when the normal handlers are not performing correctly. The debugger can replace the default system last-chance handler with its own.

In some frame before the call to the main program, the system establishes a default catch-all condition handler that issues system messages. In a subsequent frame before the call to the main program, the system usually establishes a traceback handler. These system-supplied condition handlers use condition_value to get the message and then use the remainder of the argument list to format and output the message through the system service, SYS$PUTMSG.

If the severity field of the condition_value (bits 0 through 2) does not indicate a severe_error (that is, a value of 4) these default condition handlers return with SS$_CONTINUE. If the severity is severe_error, these default handlers exit the program image with the condition value as the final image status.

The stack search ends when the old FP is 0 or is not accessible, or when 64K frames have been examined. If no condition handler is found, or all handlers returned with a SS$_RESIGNAL, then the vectored last-chance handler is called.

If a handler returns SS$_CONTINUE, and LIB$STOP was not called, control returns to the signaler. Otherwise LIB$STOP issues a message that an attempt was made to continue from a noncontinuable exception and exits with the condition value as the final image status.

Table C-1 lists all combinations of interaction between condition handler actions, the default condition handlers, the type of signals, and the call to signal or stop. In the table, "cannot continue" indicates an error which results in the message: IMPROPERLY HANDLED CONDITION, ATTEMPT TO CONTINUE FROM STOP.

Table C-1 Interaction between Handlers and Default Handlers

CALL TO:	SIGNALED CONDITION SEVERITY <2:0>	DEFAULT HANDLER GETS CONTROL	HANDLER SPECIFIES CONTINUE	HANDLER SPECIFIES UNWIND	NO HANDLER IS FOUND (STACK BAD)
LIB$SIGNAL OR HARDWARE EXCEPTION	<4	CONDITION MESSAGE RET	RET	UNWIND	CALL LAST CHANCE HANDLER EXIT
	=4	CONDITION MESSAGE EXIT	RET	UNWIND	CALL LAST CHANCE HANDLER EXIT
LIB$STOP	FORCE (=4)	CONDITION MESSAGE EXIT	"CANNOT CONTINUE" EXIT	UNWIND	CALL LAST CHANCE HANDLER EXIT

11. PROPERTIES OF CONDITION HANDLERS

Condition Handler Parameters and Invocation

If a condition handler is found on a software detected exception, the handler is called with an argument list consisting of:

continue = handler (signal_args, mechanism_args)

Each argument is a reference to a longword vector. The first longword of each vector is the number of remaining longwords in the vector. The symbols CHF$L_SIGARGLST (=4) and CHF$L_MCHARGLST (=8) can be used to access the condition handler arguments relative to AP.

Signal_args is the condition argument list from the call to LIB$SIGNAL or LIB$STOP expanded to include the PC and PSL of the next instruction to execute on a continue. In particular, the second longword is the condition_value being signaled.

Because bits 0 through 2 of the condition_value indicate severity and do not indicate which condition is being signaled, the handler should examine only the condition identification, that is, condition value (bits 3 through 27). The setting of bits 0 through 2 varies depending upon the environment. In fact, some handlers may simply change the severity of a condition and resignal. The symbols CHF$L_SIG_ARGS (=0) and CHF$L_SIG_NAME (=4) can be used to refer to the elements of the signal vectors.

Mechanism_args is a five-longword vector:

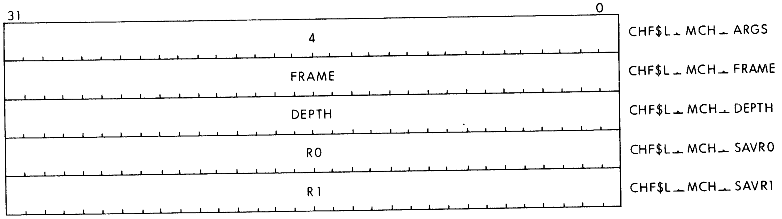


Figure C-17 Mechanism_args

The frame is the contents of the FP in the establisher's context. This can be used as a base to access the local storage of the establisher if the restrictions described in Section 11.2 are met.

The depth is a positive count of the number of procedure activation stack frames between the frame in which the exception occurred and the frame depth that established the handler being called. Depth has the value 0 for an exception handled by the procedure activation invoking the exception (that is, containing the instruction causing the hardware exception or calling LIB$SIGNAL). Depth has positive values for procedure activations calling the one having the exception (1 for the immediate caller, etc.).

If a system service gives an exception, the immediate caller of the service is notified at depth = 1. Depth has value -2 when the condition handler is established by the primary exception vector, -1 when it is established by the secondary vector, and -3 when it is established by the last-chance vector.

The contents of R0 and R1 are the same as at the time of the call to LIB$SIGNAL or LIB$STOP.

For hardware detected exceptions, the condition-value indicates which exception vector was taken and the next 0 or several longwords are additional parameters. The remaining two longwords are the PC and PSL:

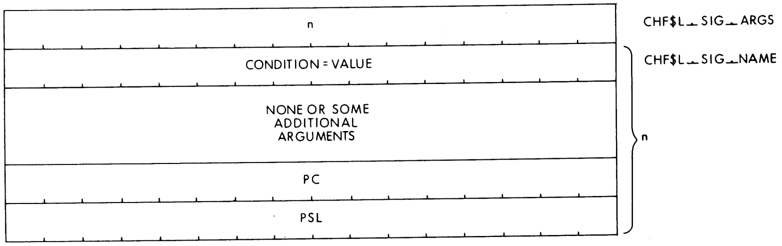


Figure C-18 Hardware Detected Exceptions

If one of the default condition handlers established by the system is entered, it calls the system service, SYS$PUTMSG, to interpret the signal argument list and output the indicated information or error message. See the description of SYS$PUTMSG in the **VAX/VMS Systems Services Reference Manual** for the format of the signal argument list.

11.2 Use of Memory

A condition handler and procedures it calls are restricted to referring to explicitly passed arguments only. Handlers cannot refer to COMMON or other external storage, and they cannot reference local storage in the procedure that established the handler. The existence of handlers does not affect compiler optimization. Compilers that do not follow this rule must ensure that any variables referred to by the handler are always in memory.

11.3 Returning from a Condition Handler

Condition handlers are invoked by the VAX Condition Handling Facility. Therefore, the return from the condition handler is to the condition handling facility.

To continue from the instruction following the signal, the handler must return with the function value SS$_CONTINUE ("true," that is, with bit 0 set). If, however, the condition was signaled with a call to LIB$STOP, the image will exit. To resignal the condition, the condition handler returns with the function value SS$_RESIGNAL ("false," that is, with bit 0 clear). To alter the severity of the signal, the handler modifies the low-order three bits of the condition-value longword in the signal-args vector and resignals. If the condition handler wants to alter the defined control bits of the signal, the handler modifies bits 31:28 of condition-value and resignals. To unwind, the handler calls SYS$UNWIND and then returns. In this case, the handler function value is ignored.

11.4 Request to Unwind

To unwind, the handler or any procedure it calls can perform:

```
success = SYS$UNWIND
          ( [depadr     = handler depth + 1],
            [new_PC     = return PC ])
```

The argument depadr specifies the address of a longword that contains the number of presignal frames (depth) to be removed. If that number is less than or equal to 0 then nothing is to be unwound. The default (address=0) is to return to the caller of the procedure that established the handler that issued the $UNWIND service. To unwind to the establisher, the depth from the call to the handler should be specified. When the handler is at depth 0, it can achieve the equivalent of an unwind operation to an arbitrary place in its establisher by altering the PC in its signal-args vector and returning with SS$_CONTINUE instead of performing an unwind.

The argument new_PC specifies the location to receive control when the unwinding operation is complete. The default is to continue at the instruction following the call to the last procedure activation removed from the stack.

The function value SUCCESS is a standard success code (SS$_NORMAL), or indicates failure with one of the following return status condition values:

- No signal active (SS$_NOSIGNAL)
- Already unwinding (SS$_UNWINDING)
- Insufficient frames for depth (SS$_INSFRAME)

The unwinding operation occurs when the handler returns to the condition handling facility. Unwinding is done by scanning back through the stack and calling each handler that has been associated with a frame. The handler is called with exception SS$_UNWIND to perform any application specific cleanup. In particular, if the depth specified includes unwinding the establisher's frame, the current handler will be recalled with this unwind exception.

The call to the handler takes the same form as previously described, with the following values:

signal_args

1

condition_value = SS$_UNWIND

mechanism_args

4

frame	establisher's frame
depth	0 (that is, unwinding self)
R0	R0 that unwind will restore
R1	R1 that unwind will restore

After each handler is called, the stack is cut back to the previous frame.

Note that the exception vectors are not checked because they are not being removed. Any function value from the handler is ignored. To specify the value of the top level "function" being unwound, the handler should modify R0 and R1 in the mechanism_args vector. They will be restored from the mechanism_args vector at the end of the unwind. Depending on the arguments to SYS$UNWIND, the unwinding operation will be terminated as follows:

- SYS$UNWIND(0,0) — unwind to the establisher's caller with the establisher function value restored from R0 and R1 in the mechanism-args vector.
- SYS$UNWIND(depth,0) — unwind to the establisher at the point of the call that resulted in the exception. The contents of R0 and R1 are restored from R0 and R1 in the mechanism_args vector.
- SYS$UNWIND(depth,location) — unwind to the specified procedure activation and transfer to a specified location with the contents of R0 and R1 from R1 in the mechanism_args vector.

SYS$UNWIND can be called whether the condition was a software exception signaled by calling LIB$SIGNAL or LIB$STOP, or was a hardware exception. Calling SYS$UNWIND is the only way to continue execution after a call to LIB$STOP.

11.5 Signaler's Registers

Because the handler is called, and can in turn call routines, the actual values of the registers that were in use at the time of the signal or exception can be scattered on the stack. To find the registers R2 through FP, a scan of stack frames must be performed starting with the current frame and ending with the call to the handler. During the scan, the last frame found to save a register contains that register's contents at the time of the exception. If no frame saved the register, the register is still active in the current procedure. The frame of the call to the handler can be identified by the return address of SYS$CALL_HANDL+4. Thus, the registers are:

R0, R1	In mechanism_args
R2..R11	Last frame saving it
AP	old AP of SYS$CALL_HANDL+4 frame
FP	old FP of SYS$CALL_HANDL+4 frame
SP	equal to end of signal-args vector+4
PC, PSL	at end of signal-args vector

12. MULTIPLE ACTIVE SIGNALS

A signal is said to be active until the signaler gets control again or is unwound. A signal can occur while a condition handler or a procedure it has called is executing in response to a previous signal. For example, procedure (A, B, C, ...) establish condition handlers (Ah, Bh, Ch, ...) respectively. If A calls B and B calls C which signals S and Ch resignals, then Bh gets control. If Bh calls procedure X and X calls procedure Y and Y signals T the stack is:

<signal T>
Y
X
Bh
<signal S>
C
B
A

which was programmed:

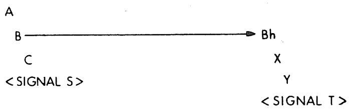


Figure C-19

The handlers are searched for in the order: Yh, Xh, Bhh, Ah. Note that Ch is not called because it is a structural descendant of B. Bh is not called again because that would require it to be recursive. Recursive handlers could not be coded in nonrecursive languages such as FORTRAN. Instead, Bh can establish itself or another procedure as its handler (Bhh).

The following algorithm is used on the second and subsequent signals which occur before the handler for the original signal returns to the condition handling facility. The primary and secondary exception vectors are checked. Then, however, the search backward in the process stack is modified. In effect, the stack frames traversed in the first search are skipped over in the second search. Thus, the stack frame preceding the first condition handler up to and including the frame of the procedure that has established the handler is skipped. Despite this skipping, depth is not incremented. The stack frames traversed in the first and second search are skipped over in a third search, etc. Note that if a condition handler signals, it will not automatically be invoked recursively. However, if a handler itself establishes a handler, this second handler will be invoked. Thus, a recursive·condition handler should start by establishing itself. Any procedures invoked by the handler are treated in the normal way that is, exception signaling follows the stack up to the condition handler.

If an unwinding operation is requested while multiple signals are active, all the intermediate handlers are called for the operation. For example, in the above diagram, if Ah specifies unwinding to A, the following handlers will be called for the unwind: Yh, Xh, Bhh, Ch, and Bh.

For proper hierarchical operation, an exception that occurs during execution of a condition handler established in an exception vector should be handled by that handler rather than propagating up the activation stack. To prevent such propagation, the vectored condition handler should establish a handler in its stack frame to handle all exceptions.

13. CHANGE HISTORY

For a change history of the Procedure Calling and Condition Handling Standard please see the VAX/VMS Run Time Library Reference Manual.

APPENDIX D
PROGRAMMING EXAMPLES

PURPOSE

The purpose of the programming examples is to illustrate VAX capabilities which are not present in the PDP-11. It is not intended to be tutorial on programming; a familiarity with PDP-11 assembly language programming is assumed.

SORT ALGORITHM

The following subroutine written in FORTRAN is an algorithm for sorting an array of values into ascending order.

```
          SUBROUTINE SORT (N, A)
          <data type x> A (N), TEMP
          INTEGER*4 N, I, J
          DO 10 I = 1, N − 1
          DO 10 J = I + 1, N
          IF (A (I) .LE.A (J)) GO TO 10
          TEMP = A (I)
          A (I) = A (J)
          A (J) = TEMP
    10    CONTINUE
          RETURN
          END
```

The following is VAX code to implement this algorithm. There is no suggestion that any given FORTRAN compiler would generate this code; the algorithm was expressed in FORTRAN only for convenience.

The subroutine is assumed to be called by the VAX standard calling convention; hence, 4 (AP) points to the address of N and 8 (AP) points to the address of A (0 origin assumed).

```
       SORT::
1.                          .WORD             (up arrow)X400C
;Entry mask to save

;R3, R2

;and enable integer

;overflow
```

```
2.                        MOVAL        @8 (AP), R0
;Get A base
3.                        MOVL         @4 (AP), R12
;Get N (size)
4.                        MOVL         #1, R1
;Initialize I
5.      1$:               ADDL3        #1, R1, R2
;Initialize J to I+1
6.      2$:               CMPx          (R0) [R1], (R0)
[R2]
;Correct order?
7.                        BLEQ         10$
;Yes
8.                        MOVx         (R0) [R1], R3
;Save A (I)
9.                        MOVx          (R0) [R2], (R0)
[R1]
;Replace A (I) with

;A (J)
10.                       MOVx         R3, (R0) [R2]
;Replace A (J) with

;saved A (I)
11.     10$:              AOBLEQ       R12, R2, 2$
;Continue
12.                       AOBLSS       R12, R1, 1$
;Continue
13.                       RET
;Return and restore

;registers R2 and R3
```

Line 1 contains an entry mask so that registers R2 and R3 will be saved by the CALL instruction which calls the subrountine. By convention, R0 and R1 are not saved. Integer overflow is enabled.

Line 2 gets the base of the A array. The move address instruction is used in conjunction with argument mode addressing. This instruction saves memory accesses inside the loop.

Line 3 gets the array size. The move long instruction is used in conjunction with argument mode addressing. This instruction saves memory accesses inside the loop.

Line 4 initializes I to 1. Literal mode addressing is used.

Line 5 initializes J with I + 1. A three operand add is used.

Line 6 compares A (I) to A (J). Register post-indexed mode addressing is used.

Line 7 branches past the exchange if the array elements are in the right order.

Lines 8 through 10 exchange the array elements if they are in the wrong order. Register post-indexed mode addressing is used.

Lines 11 and 12 carry out the loop end operations. Argument mode addressing is used.

Line 13 returns and restores registers R2 and R3.

Note, that because of logical indexing in Lines 5, 7, 8, and 9 and the orthogonality of operator and data type, the subroutine words for byte, word, longword, floating or double data types of array A simply by substituting B, W, L, F, or D respectively for x. Note that if double, then R4 would have to be saved also in the entry mask.

The size of each instruction is:

1.	2 bytes
2.	4
3.	4
4.	3
5.	4
6.	5
7.	2
8.	4
9.	5
10.	4
11.	4
12.	4
13.	1
Total	46 bytes

SIN FUNCTION

This example shows how the initial argument handling might be done in the math library to handle argument range reduction followed by CASEing to the algorithm for each octant.

```
;
; X = SIN (Y)
;
```

```
PIHI  =  xxx                                        ;high  4  bytes  (8  if
double)
PILO = xxx                                          ;low byte of 4/PI

SIN::
                               .WORD                (up arrow)X400C
;save R2-R3 for POLYF,

—R7 for POLYD

;enable integer overflow
                               MOVAL                HANDLER, 0 (FP)
;enable integer overflow

;condition handler to catch

;loss of significance on

;a huge argument
                               EMODx                #PIHI, #PILO, @4
(AP), R2, R0

;get octant in R2

;reduced argument in R0
                               BGEQ                 1$
;if positive, ok
                               ADDx                 #(up arrow)Fl. 0, R0
;if negative, get
                               DECL                 R2
;positive reduction
1$:                            BICB2                #(up arrow)C7, R2
;mask to 8 octants
                               CASEB                R2, #1, #6
;branch to each octant
2$:                            .WORD                OCT_1-2$
                               .WORD                OCT_2-2$
                               .WORD                OCT_3-2$
                               .WORD                OCT_4-2$
                               .WORD                OCT_5-2$
                               .WORD                OCT_6-2$
                               .WORD                OCT_7-2$

;fall out of CASE on octant 0
```

```
;
;octant 0 with fully precise reduced argument in R0
;
OCT_0:                        POLYx              R0, 2$, 1$
;evaluate polynomial
                              RET
;return value in R0

1$:                           .FLOAT             ...
                              .FLOAT             ...
                              ...
2$ =.−1$−1

                              ...
HANDLER:
;condition
                              .WORD              ...
                              ...
```

FIXED FORMAT FLOATING OUTPUT

This example shows how to output a floating point number in the FORTRAN format F9.3.

```
;
;string = FOUT (X)
;

STRING: .BLKB   10            ;room for output

PATTERN:                                ;EDITPC pattern string
       EO$FLOAT   4           ;float sign, move 4 digits
       EO$SEND_FLOAT                    ;end floating sign
       EO$MOVE    1           ;move one digit
       EO$INSERT  (up arrow)A/./        ;insert period
       EO$MOVE    3           ;move three fractional digits
       EO$END                 ;end of pattern
FOUT::
       .WORD      (up arrow)XC03C       ;save R2-R5, enable
                              ;overflows
       SUBL2      #8, SP                ;make room on stack
       MULF3      #(up arrow)F1000.0,@4 (AP),
R0
                              ;normalize for the .3
       CVTRFL     R0, R0                ;round digits
       CVTLP      R0, #8, (SP)          ;convert to digits on stack
       EDITPC     #8, (SP), PATTERN, STRING
```

```
                                  ;edit to output
          MOVQ          #<.LONG 9, STRING +1>, R0

                                  ;function value is a
                                  ;string descriptor
          RET                     ;return restoring R2-R5
                                  ;and the stack
```

COBOL OUTPUT EDITING

In all of these examples, A is a COMP-3 datum of length A_LEN. The operation is

```
    MOVE A TO B.
```

The generated code is

```
    EDITPC #A_LEN,@,MICRO, @B
```

In the patterns, the EO$ADJUST_INPUT can be omitted if A is the same size as B, and the EO$REPLACE_SIGN (and its EO$LOAD_FILL) can be omitted if A cannot contain a −0.

```
PICTURE                   $$,$$9.99CR

MICRO:                    EO$ADJUST_INPUT       6
                          EO$LOAD_SIGN          '$
                          EO$FLOAT              1
                          EO$INSERT             ',
                          EO$FLOAT              2
                          EO$END_FLOAT
                          EO$MOVE               1
                          EO$INSERT             '.
                          EO$MOVE               2
                          EO$LOAD_PLUS          ',
                          EO$LOAD_MINUS         'C
                          EO$STORE_SIGN
                          EO$LOAD_MINUS         'R
                          EO$STORE_SIGN
                          EO$REPLACE_SIGN       2
                          EO$REPLACE_SIGN       1
                          EO$END
PICTURE                   +$99,999.99

MICRO:                    EO$ADJUST_INPUT
                          EO$LOAD_PLUS
                          EO$STORE_SIGN
```

```
                    EO$SET_SIGNIF          7+
                    EO$INSERT              '$
                    EO$MOVE                2
                    EO$INSERT              ',
                    EO$MOVE                3
                    EO$INSERT              '.
                    EO$MOVE                2
                    EO$LOAD_FILL           '+
                    EO$REPLACE_SIGN        11
                    EO$END
PICTURE             ZZ,ZZZ.ZZ

MICRO:              EO$ADJUST_INPUT
                    EO$MOVE                2
                    EO$INSERT              ',
                    EO$MOVE                3
                    EO$SET_SIGNIF
                    EO$INSERT              '.
                    EO$MOVE                2
                    EO$BLANK_ZERO          3
                    EO$END
PICTURE             99,999.99 BLANK WHEN
                    ZERO
MICRO:
                    EO$ADJUST_INPUT        7
                    EO$SET_SIGNIF
                    EO$MOVE                2
                    EO$INSERT              ',
                    EO$MOVE                3
                    EO$INSERT              '.
                    EO$MOVE                2
                    EO$BLANK_ZERO          9
                    EO$END
PICTURE             ——————9.99

MICRO:              EO$ADJUST_INPUT        7
                    EO$FLOAT               4
                    EO$END_FLOAT
                    EO$MOVE                1
                    EO$INSERT              '.
                    EO$MOVE                2
                    EO$REPLACE_SIGN        5
                    EO$END
```

```
PICTURE             + + + + +9.99

MICRO:              EO$ADJUST_INPUT         7
                    EO$LOAD_PLUS            '+
                    EO$FLOAT                4
                    EO$END_FLOAT
                    EO$MOVE                 1
                    EO$INSERT               '.
                    EO$MOVE                 2
                    EO$LOAD_FILL            '+
                    EO$REPLACE_SIGN         5
                    EO$END

PICTURE             **,***.**

MICRO:              EO$ADJUST_INPUT         7
                    EO$LOAD_FILL            '*
                    EO$MOVE                 2
                    EO$INSERT               ',
                    EO$MOVE                 3
                    EO$SET_SIGNIF
                    EO$INSERT               ',
                    EO$MOVE                 2
                    EO$BLANK_ZERO           2
                    EO$END

PICTURE             BBBZZBZZZ.ZZB

MICRO:              EO$ADJUST_INPUT         7
                    EO$FILL                 3
                    EO$MOVE                 2
                    EO$FILL                 1
                    EO$MOVE                 3
                    EO$SET_SIGNIF
                    EO$INSERT               '.
                    EO$MOVE                 2
                    EO$BLANK_ZERO           3
                    EO$FILL                 1
                    EO$END
```

FORTRAN STATEMENT EVALUATION
FORTRAN
Assembly
Code:

```
                    J = A*K + B(1)

MOVL, R1            ;Move 1 to R1
```

```
CVTLF K, R0           ;Convert integer K to floating point
MULF2 A, R0           ;Multiply A*K and store in R0
ADDF2 B               ;Add B indexed by R1 to R0
[R1], R0
CVTFL R0, J           ;Convert result in R0 to integer and
                      store in J
```

This program evaluates the FORTRAN statement listed above. I is a subscript which moved to register R1. The next step of the program converts the integer K to a floating point number. Next A is multiplied by K and the result is stored in register R0. The value I, which is stored in register R1, indexes B and the calculated result is added to R0 which currently contains A*K. The last step of the program converts the floating point result back to integer format, and stores the integer in location J.

VARIABLE LENGTH FIELD
PL1 Assembly

```
Code: DECLARE A (1:10) BIT      ;Vector A, elements 1-10,
(5)                             ;5 bit field

A(I) = A(I) + 1                 ;Increment Ith element of
                                ;A and store in a
```

Machine Code:

```
INDEX I, #1, #10, #5            ;Calculate index
#—5, R0

EXTV R0, #5, A, R1              ;Extract 5 bits and store
                                ;in R1

INCL R1                         ;Increment R1

INSV R1, R0, #5, A              ;Store 5 bits into A
                                ;offset by R0
```

This example shows the use of the variable length field instructions using the PL1 Programming Language. Its purpose is to add 1 to a particular field within a vector of fields. In the assembler code, the

DECLARE statement informs the compiler that A is a vector, its elements are numbered 1 through 10, and each element is a field five bits wide. The A(I) statement increments the Ith element of A and stores the result back in A.

In the machine code, the INDEX statement consists of a lower limit of 1, an upper limit of 10, field size of 5, an offset of −5, and a temporary (R0) to store the result of the index calculation. The offset of −5 is required since the subscript starts at 1 but all indexing starts at 0.

The INDEX statement in this example checks I in the range from 1 through 10. If I is in this range it is multiplied by the field size of 5, the offset of −5 is added, and the result is stored in R0. Thus, R0 will contain the position offset of the field A(I) from the start of A. If I is outside the range 1 through 10, a subscript range trap occurs and typically results in an error message.

LOOPS

FORTRAN:

```
INTEGER *2 L                    ;Use L as a word for a
DO 1 L = 3, 10, 2               ;loop counter—L is
                                ;an integer of 2 bytes
                                ;and loop is incremented
                                ;by 2 for each pass
                                ;through loop
1 CONTINUE
```

Assembly

```
Code:       MOVW #3, L
START:      ACBW #10, #2, L, START
```

FORTRAN:

```
INTEGER LL                      ;Use LL as a word for a
DO 1 LL = 1, 10                 ;loop counter. Loop is
                                ;incremented by one for
                                ;each pass through loop.
1 CONTINUE
```

Assembly

```
Code:       MOVL #1, LL
START:      AOBLEQ #10, LL, START
```

APPENDIX E
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 access type:

a — Calculate the effective address of the specified operand. Address is returned in a longword which is the actual instruction operand. Context of address calculation is given by <data type>.

b — No operand reference. Operand specifier is a branch displacement. Size of branch displacement is given by <data type>.

m — Operand is read, potentially modified and written. Note that this is NOT an indivisible memory operation. Also note that if the operand is not actually modified, it may not be written back. However, modify type operands are always checked for both read and write accessibility.

r — Operand is read only.

v — Calculate the effective address of the specified operand. If the effective address is in memory, the address is returned in a longword which is the actual instruction operand. Context of address calculation is given by <data type>.

If the effective address is 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	D_floating
f	F_floating
g	G_floating
h	H_floating
l	Longword
o	octaword
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
–	subtraction, 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 Longword
(x)	contents of memory location whose address is x
(x)+	contents of memory location whose address is x; x incremented by the size of operand referenced at x

−(x)	x decremented by size of operand to be referenced at x; contents of memory location whose address is x
<x:y>	a modifier which delimits an extent from bit position x to bit position y inclusive
<x1,x2,...,xn>	a modifier which enumerates bits x1,x2,...,xn
x...y	x through y inclusive
{ }	braces used to indicate precedence
AND	logical AND
OR	logical OR
XOR	logical XOR
NOT	logical (1's) complement
LSS	less than signed
LSSU	less than unsigned
LEQ	less than or equal signed
LEQU	less than or equal unsigned
EQL	equal signed
EQLU	equal unsigned
NEQ	not equal signed
NEQU	not equal unsigned
GEQ	greater than or equal signed
GEQU	greater than or equal unsigned
GTR	greater than signed
GTRU	greater than unsigned
SEXT (x)	x is signed-extended to size of operand needed
ZEXT (x)	x is zero-extended to size of operand needed
REM (x, y)	remainder of x divided by y, such that x/y and REM (x,y) have the same sign
MINU (x, y)	minimum unsigned of x and y
MAXU (x, y)	maximum unsigned of x and y

The following conventions are used:

- Other than that caused by (   ) +, or −(   ), and the advancement of PC, only operands or portions of operands appearing on the left side of assignment statements are affected.
- No operator precedence is assumed other than that replacement (←) has the lowest precedence. Precedence is indicated explicitly by {   }.
- All arithmetic, logical, and relational operators are defined in the context of their operand. For example, + applied to floating operands means a floating add while + applied to byte operands is an integer byte add. Similarly, **LSS** is a floating comparison when applied to floating operands while **LSS** is an integer byte comparison when applied to byte operands.
- Instruction operands are evaluated according to the operand specifier conventions. The order in which operands appear in the instruction description has no effect on the order of evaluation.
- Condition codes are, in general, affected on the value of actual stored results, not on "true" results (which might be generated internally to greater precision). Thus, for example, two positive integers can be added together and the sum stored, because of overflow, as a negative value. The condition codes will indicate a negative value even though the "true" result is clearly positive.

APPENDIX F
ASSEMBLER NOTATION

INTRODUCTION

The VAX assembler provides, as a subset, a notation which is very similar to the PDP-11 assembler notation. The principal differences are due to the fact that the VAX architecture has new addressing modes and has several length variations of modes for which the PDP-11 has only a single length. For example, the PDP-11 has displacement addressing with a single displacement size of 16 bits. VAX has displacement addressing in various forms with displacements of 8, 16, and 32 bits.

In general, the programmer need not be aware of the length variations in VAX addressing modes. He or she simply writes the addressing mode in a format identical to the analogous PDP-11 addressing mode, and the assembler will choose the shortest form of addressing consistent with the state of symbol definition at assembly time. Occasionally, a programmer may wish to force a given length addressing mode; the VAX assembler includes a notation for accomplishing this. (Of course, if the programmer forces a length which cannot be accomodated at assembly or link time, the assembler or linker will generate an error indication.)

NOTATION FOR GENERAL MODE ADDRESSING

Register Mode

The general notation is Rn. Since results are unpredictable if R is PC for operands taking a single register; if Rn is SP or PC for operands taking a pair of registers; or if Rn is AP, FP, SP, or PC for operands taking four registers, the assembler generates an error indication.

Register Deferred Mode

The general notation is (Rn). Since results are unpredictable if Rn is PC, the assembler generates an error indication.

Autoincrement Mode

The general notation is (Rn)+.

Autoincrement Deferred Mode

The general notation is @(Rn)+.

Autodecrement Mode

The general notation is −(Rn). Since results are undefined if R is PC the assembler generates an error indication.

Displacement Mode

The general notation is D(Rn). To force a byte, word, or longword displacement, the notation is B↑D(Rn), W↑D(Rn), L↑D(Rn), respectively. If a general address G is used, the assembler assembles this as D(PC) where D = G −{updated value of PC}. This latter form is termed PC-relative addressing. If a form is forced which is shorter than the actually needed displacement, the assembler or linker generates an error indication.

Displacement Deferred Mode

The general notation is @D(Rn). To force a byte, word, or longword displacement, the notation is @B↑D(Rn), @W↑D(Rn), @L↑D(Rn) respectively. If a general address @G is used, the assembler assembles this as @D(PC) where D = G − {updated value of PC}. This latter form is termed PC-relative deferred addressing. If a form is forced which is shorter than the actually needed displacement, the assembler or linker generates an error indication.

Literal Mode

The general notation is #cons. Depending on the value of the constant, this results in either immediate or literal mode. To force literal mode, the notation is S↑#cons. To force immediate mode, the notation is I↑# cons. If either literal or immediate mode is used on a modify or write operand, the assembler generates an error indication.

Absolute Addressing Mode

To force a reference to an absolute address the notation is @#location. This is assembled as autoincrement deferred using PC.

General Addressing

When a reference to a symbol will be either absolute or PCrelative, but the choice is to be determined by the linker, the notation is G↑location. This is assembled as a five-byte operand. The linker chooses either @#location or L↑D(PC) depending on whether the location is absolute or PC-relative. This is used both for general external references and for general references between program sections (PSECTs).

Index Mode

The general notation is <base operand mode>[Rx] where <base operand mode> is the notation for any of the addressing modes register deferred, autoincrement (immediate), autoincrement deferred (absolute), autodecrement, displacement (PC-relative), displacement deferred (PC-relative deferred), or general addressing. Since the result is unpredictable if a register in the base operand mode is the same as the index register (except for PC), the assembler generates an error.

GENERAL MODE ADDRESSING SUMMARY

Symbolic	Assembled Mode
1. R	register
2. (R)	register deferred
3. (R)+	autoincrement
4. −(R)	autodecrement
5. D(R)	byte, word, or longword displacement; register deferred; default is word if D is not known
6. B↑D(R)	byte displacement
7. W↑D(R)	word displacement
8. L↑D(R)	longword displacement
9. G	byte, word, or longword displacement off PC default is longword if G is not known
10. B↑G	byte displacement off PC
11. W↑G	word displacement off PC
12. L↑G	longword displacement off PC
13. G↑G	general addressing (absolute or PC-relative)
14. #cons	autoincrement of PC (immediate) or literal
15. S↑#cons	short literal
16. I↑#cons	immediate
17. (R)[Rx]	register deferred indexed
18. (R)+[Rx]	autoincrement indexed
19. #cons[Rx]	autoincrement of PC (immediate) indexed
20. I↑#cons[Rx]	autoincrement of PC (immediate)indexed
21. −(R)[Rx]	autodecrement indexed
22. D(R)[Rx]	byte, word, or longword displacement indexed; register deferred indexed
23. B↑D(R)[Rx]	byte displacement indexed
24. W↑D(R)[Rx]	word displacement indexed

Symbolic	Assembled Mode
25. L↑D(R)[Rx]	longword displacement indexed
26. G[Rx]	byte, word, or longword displacement off PC indexed
27. B↑G[Rx]	byte displacement off PC indexed
28. W↑G[Rx]	word displacement off PC indexed
29. L↑G[Rx]	longword displacement off PC indexed
30. G↑location [Rx]	general (absolute or PC-relative) indexed
31. @(R)[Rx]	byte displacement deferred indexed with 0 displacement
32. @(R)+[Rx]	autoincrement deferred indexed
33. @#location[Rx]	autoincrement of PC (immediate) deferred indexed
34. @D(R)[Rx]	byte, word, or longword displacement deferred indexed
35. @B↑D(R)[Rx]	byte displacement deferred indexed
36. @W↑D(R)[Rx]	word displacement deferred indexed
37. @L↑D(R)[Rx]	longword displacement deferred indexed
38. @G[Rx]	byte, word, or longword displacement off PC deferred indexed
39. @B↑G[Rx]	byte displacement off PC deferred indexed
40. @W↑G[Rx]	word displacement off PC deferred indexed
41. @L↑G[Rx]	longword displacement off PC deferred indexed
42. @(R)	byte displacement deferred with 0 displacement
43. @(R)+	autoincrement deferred
44. @#location	autoincrement of PC (immediate)
45. @D(R)	byte, word, longword displacement deferred
46. @B↑D(R)	byte displacement deferred
47. @W↑D(R)	word displacement deferred
48. @L↑D(R)	longword displacement deferred

Symbolic	Assembled Mode
49. @G	byte, word, or longword displacement off PC deferred
50. @B↑G	byte displacement off PC deferred
51. @W↑G	word displacement off PC deferred
52. @L↑G	longword displacement off PC deferred

BRANCH DISPLACEMENT ADDRESSING

The general notation is locn, where locn is the branch address. The assembler fills in the displacement displ where displ = locn−{updated value of PC}.

GENERIC OPCODE SELECTION

As a convenience to the programmer, the assembler automatically selects from among similar instructions. This allows the programmer to write code without worrying about these distinctions.

Branch Selection

If the programmer gives BR or BSB as the mnemonic, the assembler will automatically select either BRB or BRW (BSBB or BSBW) based on the distance to the label. If the label is not yet defined, the word branch displacement form will be selected.

Number of Operand Selection

If the programmer omits the final digit from those opcodes which have two forms (e.g., ADDW instead of ADDW2 or ADDW3), the assembler will select the correct form based on the number of operands specified by the user.

APPENDIX G
OPERAND PROCESSING

The following three steps are performed in order by each instruction:

1. Each operand specifier is evaluated in order of instruction stream occurrence as follows:

access type	evaluation
read	evaluate operand location, read the operand and save the operand
write	evaluate operand location and save the location
modify	evaluate operand location, read the operand and save both location and operand
address, branch	evaluate the address and save the address
field	evaluate operand base location and save the location

2. Perform the operation indicated by the instruction.
3. Store the result(s) using the saved address in the order indicated by the occurrence of operand specifiers in the instruction stream.

NOTE

The string (character and numeric) instructions write any output strings and store the registers during step 3.

The implications of this processing are:

1. Autoincrement and autodecrement operations occur as the operand specifiers are processed, and subsequent operand specifiers use the updated contents of registers modified by those operations.

NOTE

This implication does not necessarily apply to floating point operands.

2. Except for those operations mentioned in step 1, all input operands are read and all addresses of output operands are computed before any results of the instruction are stored.

3. An operand of modify access type is not read, modified, and written as an indivisible operation. Therefore, modify access type operands cannot be used for synchronization. For synchronization refer to the ADAWI instruction, INSQUE and REMQUE instructions, and BBCCI and BBSSI instructions.
4. If an instruction references two operands of write or modify access type at the same or overlapping address, the first will be overwritten by the second. If an instruction modifies a register implicitly and also has an output operand, the output store is performed after the register update.

APPENDIX H
ACCURACY

It will now be shown that an overflow bit and two guard bits are adequate to guarantee accuracy of rounded ADD, SUB, MUL, or DIV, provided, of course, that the algorithms are properly chosen. Note, first, that ADD and SUB may result in propagation of a carry, and hence the overflow bit is necessary. Second, if in ADD or SUB there is a one-bit loss of significance in conjunction with an alignment shift of two or more bits, the first guard bit is needed for the LSB of the normalized result, and the second is then the rounding bit. So the three bits are necessary. A number of constraints must be observed in selection of the algorithms for the basic operations in order for these three bits to be sufficient to guarantee an error bound of (½) LSB:

1. ADD or SUB:
   - If the alignment shift does not exceed 2 there are no constraints, because no bits can be lost.
   - If the alignment shift exceeds 2 (or however many guard bits are used, say $g \geq 2$), no negations may be made after the aligment shift takes place.
   - If the above constraint is observed, the error bound for a rounded result is (½) LSB. If, however, a negation follows the alignment shift, the error bound will be

     (½) * (1 + 2**(−g+2)) LSB

     because a "borrow" will be lost on an implicit subtraction, if nonzero bits were lost in the alignment shift. Note that the error bound is 1 LSB if the constraint is ignored and there are only two guard bits ($g = 2$).
   - The constraint on no negations after the alignment shift may be replaced by keeping track of nonzero bits lost during the alignment shift, and then negating by one's complement if any "ones" were lost, and by two's complement if none were lost. If this is done, the error bound will be (½) LSB.
2. MUL:
   - The product of two normalized binary fractions can be as small as ¼ and must be less than one. The overflow bit is not needed for MUL, but the first guard bit will be necessary for normalization if the product if less than ½, and, in this case, the second guard bit is the rounding bit.

- The first constraint on MUL is that the product be generated from the least to the most significant bit. Low order bits, in positions to the right of the second guard bit, may be discarded, but ONLY AFTER they have made their contribution to carries which could propagate into the guard bits or beyond.
- For the same reasons as for ADD or SUB, if low order bits of the product have been discarded, no negations can be made after generating the product.

3. DIV:
   - For standard algorithms it is necessary that the remainder be generated exactly at each step; the overflow and two guard bits are adequate for this purpose. The register receiving the quotient must have a guard bit for the rounding bit, and the quotient must be developed to include the rounding bit.
   - The Newton-Raphson quadratic convergence algorithms require a number of guard bits equal to twice the number of bits desired in the result if the correctness of the rounding bit is to be guaranteed.

VAX observes all constraints and generates floating point results with an error bound of (½) LSB for all floating instructions except EMOD and POLY (see EMOD and POLY descriptions.)

GLOSSARY

abort An exception that occurs in the middle of an instruction and potentially leaves the registers and memory in an indeterminate state, such that the instruction can not necessarily be restarted.

absolute indexed mode An indexed addressing mode in which the base operand specifier is addressed in absolute mode.

absolute mode Autoincrement deferred mode in which the PC is used as the register. The PC contains the address of the location containing the actual operand.

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. 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 users 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 Program Counter (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.

alphanumeric character An upper or lower case letter (A to Z, a to z), a dollar sign ($), an underscore (_), or a decimal digit (0 to 9).

American Standard Code for Information Interchange (ASCII) A set of 8-bit binary numbers representing the alphabet, punctuation, numerals, control, and other special symbols used in text representation and communications protocol.

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.

autodecrement index mode An indexed addressing mode in which the base operand specifier users 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 of 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 F_floating, 8 for quadword, G_floating and D_floating, and 16 for octaword and H_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 register A general register used to contain the address of the first entry in a list, table, array, or other data structure.

bit complement (also called *one's complement*) The result of exchanging 0s and 1s in the binary representation of a number. Thus, the bit complement of the binary number 11011001 (217_{10}) is 00100110. Bit complements are used in place of their corresponding binary numbers in some arithmetic computations in computers.

bit string See *variable length bit field*

block 1. The smallest addressable unit of data that the specified device can transfer in an I/O operation (512 contiguous bytes for most disk devices) 2. An arbitrary number of contiguous bytes used to store logically related status, control, or other processing information.

branch access type An instruction attribute which indicates that the processor does not reference an operand address, but rather that the operand is a branch displacement. The size of the branch displacement is given by the data type of the operand.

branch mode In branch address mode, the instruction operand specifier is a signed byte or word displacement. The displacement is added to the contents of the updated PC (which is the address of the first byte beyond the displacement), and the result is the branch address.

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 the interrupt service context has one call stack.

character A symbol represented by an ASCII code. See also *alphanumeric character*.

character string A contiguous set of bytes. A character string is identified by two attributes: an address and a length. Its address is the address of the byte containing the first character of the string. Subsequent characters are stored in bytes of increasing addresses. The length is the number of characters in the string.

character string descriptor A quadword data structure used for passing character data (strings). The first word of the quadword contains the length of the character string. The second word can contain type information. The remaining longword contains the address of the string.

command An instruction, generally an English word, typed by the user at a terminal or included in a command file, which requests the software monitoring a terminal or reading a command file to perform some well-defined activity. For example, typing the COPY command requests the system to copy the contents of one file into another file.

command procedure A file containing commands and data that the command interpreter can accept in lieu of the user's typing the commands individually on a terminal.

compatibility mode A mode of execution that enables the central processor to execute nonprivileged PDP-11 instructions. The operating system supports compatibility mode execution by providing an RSX-11M programming environment for an RSX-11M task image. The operating system compatibility mode procedures reside in the control region of the process executing a compatibility mode image. The procedures intercept calls to the RSX-11M executive and convert them to the appropriate operating system functions.

condition An exception condition detected and declared by software.

condition codes Four bits in the Processor Status Word (PSW) 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. The operating system searches for a condition handler and, if it is found, initiates the handler immediately. The condition handler may perform some act 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.

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 hardcopy terminal. It enables the operator to start and stop the system, monitor system operation, and run diagnostics.

console terminal The hardcopy terminal connected to the central processor console.

context Sometimes also called *process state*. See *hardware context*.

context indexing The ability to index through a data structure automatically because the size of the data type is known and is used to determine the offset factor.

context switching Interrupting the activity in progress and switching to another activity. Context switching occurs as one process after another is scheduled for execution. The operating system saves the interrupted process's hardware context in its hardware process control block (PCB) using the Save Process Context instruction, then loads another process's hardware PCB into the hardware context using the Load Process Context instruction, scheduling that process for execution.

control region The highest-addressed half of 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 conrol 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.

counted string A data structure consisting of a byte-sized length followed by the string.

current access mode The processor access mode of the currently executing software. The Current Mode field of the Processor Status Longword (PSL) indicates the access mode of the currently executing software.

D_floating datum Eight contiguous bytes 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 contain 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 represents 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 D_floating datum is in the approximate range (+ or −) 0.29 × $10^{—38}$ to 1.7 × 10^{38}. The precision is approximately one part in 2^{55}, or sixteen decimal digits.

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, quadword, and octaword integer; F_floating, D_floating, G_floating, and H_floating; character string; packed decimal string;

and variable length bit field.

descriptor A data structure used in calling sequences for passing argument types, addresses and other optional information. See character string descriptor.

device interrupt An interrupt received on interrupt priority levels 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). A colon (:) separates it from following fields.

device register A location in device controller logic used to request device functions (such as I/O transfers) and/or to 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 mapping cache A cache organization in which only one address comparison is needed to locate any data in the cache because any block of main memory data can be placed in only one possible position in the cache. Contrast with *fully associative cache.*

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 registers. The result is the address of a longword which contains the address of the actual operand. If the Program Counter 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.

double floating datum See *D_floating datum*.

drive The electromechanical unit of a mass storage device system on which a recording medium (disk cartridge, disk pack, or magnetic tape reel) is mounted.

effective address The address obtained after indirect or indexing modifications are calculated.

entry mask A word whose bits represent the registers to be saved or restored on a subroutine or procedure call using the Call and Return instructions.

entry point A location that can be specified as the object of a call. It contains an entry mask and exception enables known as the *entry point mask*.

escape sequence An escape is a transition from the normal mode of operation to a mode outside the normal mode. The escape character is the code that indicates the transition from normal to escape mode. An escape sequence refers to the set of character combinations starting with an escape character that the terminal transmits without interpretation to the software set up to handle sequences.

event A change in process status or an indication of the occurrence of some activity that concerns an individual process or cooperating processes. An incident reported to the scheduler that affects a process's ability to execute. Events can be synchronous with the process's execution (e.g., a wait request), or they can be asynchronous (e.g., I/O completion). Some other events include: swapping, wake request or page fault.

event flag A bit in an event flag cluster that can be set or cleared to indicate the occurrence of the event associated with that flag. Event flags are used to synchronize activities in a process or among many processes.

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 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*.

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.

F_floating datum Four contiguous bytes starting on an addressable byte boundary. The bits are labeled from right to left 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 represents 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 × 10^{-38} to 1.7 × 10^{38}. The precision is approximately one part in 2^{23}, or seven decimal digits.

fault A hardware exception condition that occurs in the middle of an instruction and that leaves the registers and memory in a consistent state, such that elimination of the fault and restarting the instruction will give correct results.

field 1. See *variable length bit field*. 2. A set of contiguous bytes in a logical record.

floating (point) datum See *F_floating datum*.

frame pointer General register 13(R13). By convention, FP contains the base address of the most recent call frame on the stack.

fully associative cache A cache organization in which any block of data from main memory can be placed anywhere in the cache. Address comparision must take place against each block in the cache to find any particular block. Constrast with *direct mapping cache.*

G_floating datum A G_floating datum is eight contiguous bytes starting on an arbitary byte boundary. The bits are labelled from the right 0 through 63. A G_floating datum is specified by its address A, the address of the byte containing bit 0. The form of a G_floating datum is sign magnitude with bit 15 the sign bit, bits 14:4 an excess-1024 binary exponent, and bit 3:0 and 63:16 a normalized 53-bit fraction with the redundant, most significant fraction bit not represented. Within the fraction, bits of increasing significance go 48 through 63, 32 through 47, 16 through 31, and 0 through 3. The 11-bit exponent field encodes the values 0 through 2047. An exponent value of 0, together with a sign bit of 0, is taken to indicate that the G_floating datum has a value of 0. Exponent values of 1 through 2047 indicate true binary exponents of −1023 through +1023. An exponent value of 0, together with a sign bit of 1, is taken as reserved. Floating point instructions processing a reserved operand take a reserved operand fault. The value of a G_floating datum is in the approximate range 0.56×10^{-308} through 0.9×10^{-308}. The precision of a G_floating datum is approximately one part in 2^{52}, i.e., typically 15 decimal digits.

general register Any of the sixteen 32-bit registers used as the primary operands of the native mode instructions. The general registers include 12 general purpose registers which can be used as accumulators, as counters, and as pointers to locations in main memory, and the Frame Pointer (FP), Argument Pointer (AP), Stack Pointer (SP), and Program Counter (PC) registers.

giga Metric term used to represent the number 1 followed by nine 0s (10^9, though in the computer industry it is often used to mean 2^{30}, which is about 7.4% larger.)

H_floating datum An H_floating datum is 16 contiguous bytes starting on an arbitrary byte boundary. The bits are labelled from the right 0 through 127. An H_floating datum is specified by its address A, the address of the byte containing bit 0. The form of an H_floating datum is sign magnitude with bit 15 the sign bit, bits 14:0 an excess-16,384 binary exponent, and bits 127:16 a normalized 113-bit fraction with the redundant, most significant fraction bit not represented. Within the fraction, bits of increasing significance go 112 through 127, 96 through 111, 80 through 95, 64 through 79, 48 through 63, 32 through 47, and 16 through 31. The 15-bit exponent field encodes the values 0 through 32,767. An exponent value of 0 together with a sign bit of 0, is taken to indicate that the H_floating datum has a value of 0. Exponent values of 1 through 32,767 indicates true binary exponents of −16383 through + 16383. An exponent value of 0, together with a sign bit of 1, is taken as reserved. Floating point instructions processing a reserved operand take a reserved operand fault. The value of an H_floating datum is in the approximate range 0.84×10^{-4932} through 0.59×10^{-4932}. The precision of an H_floating datum is approximately one part in 2^{112}, i.e., typically 33 decimal digits.

hardware context The values contained in the following registers while a process is executing: the Program Counter (PC); the Processor Status Longword (PSL); the 14 general registers (R0 through R13); the four processor registers (P0BR, P0LR, P1BR and P1LR) that describe the process virtual address space; the Stack Pointer (SP) for the current access mode in which the processor is executing; plus the contents to be loaded in the stack pointer for every access mode other than the current access mode. While a process is executing, its hardware context is continually being updated by the processor. While a process is not executing its hardware context is stored in its hardware PCB.

hardware process conrol block (PCB) A data structure known to the processor that contains the hardware context when a process is not executing. A process's hardware PCB resides in its process header.

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

immediate mode Autoincrement mode addressing in which the PC is used as the register.

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 (called 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.

input stream The source of commands and data. One of either the user's terminal, the batch stream, or an indirect command file.

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 conrol logic continously fetches data from memory to keep the 8-byte buffer full.

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

interrecord gap A blank space deliberately placed between data records on the recording surface of a magnetic tape.

interrupt An event other than an exception or Branch, Jump, Case, or Call instruction that changes the normal flow of instruction execution. Interrupts are generally external to the process executing when the interrupt occurs. See also *device interrupt*, *software interrupt*, and *urgent interrupt*.

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). IPL 1 is lowest, 31 highest. The levels arbitrate contention for processor service. For example, a device cannot interrupt the processor if it is currently executing at an IPL greater than the one of the device's interrupt service routine.

interrupt service routine The routine executed when a device interrupt occurs.

interrupt stack The systemwide 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 systemwide 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*.

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.

literal mode In literal mode addressing, the instruction operand is a constant whose value is expressed in a 6-bit field of the instruction. If the operand data type is byte, word, longword, quadword, or octaword, the operand is zero-extended and can express values in the range 0 through 63 (decimal). If the operand data type is F_floating, D_floating, G_floating, or H_floating, the 6-bit field is composed of two 3-bit fields, one for the exponent and the other for the fraction.

longword Four contiguous bytes starting on an addressable byte boundary. Bits are numbered from right to left 0 through 31. The address of the longword is the address of the byte containing bit 0. When interpreted arithmetically, a longword is a 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, the value is in the range 0 through 4,294,967,295.

main memory See *physical memory.*

mass storage device A device capable of reading and writing data on mass storage media such as a diskpack or a magnetic tape reel.

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.

native mode 1) The processor's primary execution mode. In it, programmed instructions are interpreted as byte-aligned, variable length instructions that operate on byte, word, longword, quadword, and octaword integer, F_floating, D_floating, G_floating, and H_floating, character string, packed decimal, and variable length bit field data. 2) The instruction execution mode other than compatibility mode.

nibble The low-order or high-order four bits of an 8-bit byte.

normalized fraction A numeric representation patterned on scientific notation, but in which the fraction part of the representation is greater than or equal to 0.5 and less than 1. As a binary form, such a fraction will always begin with a 1 in the leftmost (most significant) bit, unless the number is zero. Because of this, the lead 1 is not stored, and a bit-per-number saving is effected in storage.

numeric string A contiguous sequence of bytes representing up to 31 decimal digits (one per byte) and possily a sign. The numeric string is specified by its lowest addressed location, its length, and its sign representation.

octaword An octaword is 16 contiguous bytes starting on an arbitrary byte boundary. The bits are numbered from the right 0 through 127. An octaword is specified by its address A, the address of the byte containing bit 0. When interpreted arithmetically, an octaword is a two's complement integer with bits of increasing significance going 0 through 126 and bit 127 the sign bit. The value of the integer is in the range -2^{127} to $2^{127}-1$. The octaword data type is not yet fully supported by VAX instructions.

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.

one's complement See *bit complement.*

opcode The pattern of bits within an instruction that specifies 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.

packed decimal A method of representing a decimal number by storing a pair of decimal digits in one byte, taking advantage of the fact that only four bits are required to represent the numbers 0 through 9.

packed decimal string A contiguous sequence of up to 16 bytes interpreted as a string of nibbles. Each nibble represents a digit, except the low-order nibble of the highest addressed byte, which represents the sign. The packed decimal string is specified by its lowest addressed location and the number of digits.

page 1. A set of 512 contiguous byte locations used as the unit of memory mapping and protection. 2. The data between the beginning of file and a page marker, between two markers, or between a marker and the end of a file.

page fault An exception generated by a reference to a page which is not mapped into a working set.

page fault cluster size The number of pages read in on a page fault.

page frame number (PFN) The address of the first byte of a page in physical memory. The high-order 21 bits of the physical address of the base of a page.

page 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. When it is not in memory, the PTE 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, that is, one process cannot exceed the working set limit assigned to it by bringing in more than its quota of pages. This protects other processes in the system.

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 memory The memory modules connected to the synchronous backplane interconnect 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.

privileged instructions In general, any instruction intended for use by the operating system or privileged system programs. In particular, instructions that the processor will not execute unless the current access mode is kernel mode (e.g., HALT, SVPCTX, LDPCTX, MTPR, and MFPR).

procedure 1. A routine entered via a Call instruction. 2. See *command procedure*.

process The basic entity scheduled by the system software, that provides the context in which an image executes. A process consists of an address space and both hardware and software contexts.

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 the 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 page tables The page tables used to describe process virtual memory.

process space The lowest-addressed half of virtual address space, where 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.

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 n. A circular, doubly-linked list. v. To make an entry in a list or table, perhaps using the INSQUE instruction.

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

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.

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.

signal 1. An electrical impulse conveying information. 2. The software mechanism used to indicate that an exception condition was detected.

software interrupt An interrupt generated on interrupt priority levels 1 through 15, which can be requested only by software.

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 and going 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).

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 Backplane Interconnect (SBI) That part of the hardware that interconnects the processor, memory controllers, MASSBUS adaptors, and the UNIBUS adaptor.

system In the context "system, owner, group, world," *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 which contains 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 identification register A processor register which contains the processor type and serial number.

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 system base register (SBR).

system region The third quarter of virtual address space, i.e., the lower-addressed half of system space. Virtual addresses in the system region are shareable between processes. Some of the data structures mapped by system region virtual addresses are system entry vectors, the system control block (SCB), the system page table (SPT), and process page tables.

system space The higher-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 LA38 DECwriter hard-copy terminal and VT100 video display terminal.

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

trap An exception condition that occurs at the end of the instruction that caused the exception. The PC saved on the stack is the address of the next instruction that would normally have been executed. All software can enable and disable some of the trap conditions with a single instruction.

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 either 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, and it takes advantage of the features of both.

unit record device A device such as a card reader or lineprinter.

unwind the call stack To remove call frames from the stack by tracing back through nested procedures 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 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.

value return registers The general registers R0 and R1 used by convention to return function values. These registers are not preserved by any called procedures. They are available as temporary registers to any called procedure. All other registers (R2, R3,..., R11, AP, FP, SP, PC) are preserved across procedure calls.

variable length bit field A set of 0 to 32 contiguous bits located arbitrarily with respect to byte boundaries. A variable bit field is specified by four attributes: 1) the address A of a byte, 2) the bit position P of the starting location of the bit field with respect to bit 0 of the byte 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 idenfity 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 of data. The virtual address space seen by the programmer is a linear array of 4,294,967,296(2^{32}) byte addresses.

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 nonresident virtual memory.

virtual page number The virtual address of a page of virtual memory.

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 address space to which an executing 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.

write access type The specified operand of an instruction or procedure is only written during that instruction's execution.

write allocate A cache management technique in which cache is allocated on a write miss as well as on the usual read miss.

write back A cache management techinque 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

VAX ARCHITECTURE HANDBOOK 1981–82 **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 __________

Title __________

Company __________ Dept. __________

Address __________

City __________ State __________ Zip __________

(staple here)

(staple here)

— — — — — — — — — — — — — — (please fold here) — — — — — — — — — — — — — —

No Postage
Necessary
if Mailed in the
United States

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 33 MAYNARD, MASS.

POSTAGE WILL BE PAID BY ADDRESSEE

DIGITAL EQUIPMENT CORPORATION
NEW PRODUCTS MARKETING
PK3-1/M92
MAYNARD, MASS. 01754

digital

HANDBOOK SERIES

Microcomputers and Memories
Microcomputer Interfaces
PDP-11 Processor
PDP-11 Software
Peripherals
Terminals and Communications
VAX Architecture
VAX Software
VAX Hardware

digital

DIGITAL EQUIPMENT CORPORATION, Corporate Headquarters: Maynard, MA 01754, Tel. (617) 897-5111 — SALES AND SERVICE OFFICES; UNITED STATES — ALABAMA, Birmingham, Huntsville ARIZONA, Phoenix, Tucson ARKANSAS, Little Rock CALIFORNIA, Costa Mesa, El Segundo, Los Angeles, Oakland, Sacramento, San Diego, San Francisco, Monrovia, Santa Barbara, Santa Clara, Sherman Oaks COLORADO, Colorado Springs, Denver CONNECTICUT, Fairfield, Meriden DELAWARE, Newark FLORIDA, Miami, Orlando, Pensacola, Tampa GEORGIA, Atlanta HAWAII, Honolulu IDAHO, Boise ILLINOIS, Chicago, Peoria INDIANA, Indianapolis IOWA, Bettendorf KENTUCKY, Louisville LOUISIANA, New Orleans MARYLAND, Baltimore MASSACHUSETTS, Boston, Springfield, Waltham MICHIGAN, Detroit, Kalamazoo MINNESOTA, Minneapolis MISSOURI, Kansas City, St. Louis NEBRASKA, Omaha NEW HAMPSHIRE, Manchester NEW JERSEY, Cherry Hill, Parsippany, Princeton, Somerset NEW MEXICO, Albuquerque, Los Alamos NEW YORK, Albany, Buffalo, Long Island, New York City, Rochester, Syracuse, Westchester NORTH CAROLINA, Chapel Hill, Charlotte OHIO, Cincinnati, Cleveland, Columbus, Dayton OKLAHOMA, Tulsa OREGON, Portland PENNSYLVANIA, Harrisburg, Philadelphia, Pittsburgh RHODE ISLAND, Providence SOUTH CAROLINA, Columbia, Greenville TENNESSEE, Knoxville, Nashville TEXAS, Austin, Dallas, El Paso, Houston, San Antonio UTAH, Salt Lake City VERMONT, Burlington VIRGINIA, Fairfax, Richmond WASHINGTON, Seattle, Spokane WASHINGTON D.C. WEST VIRGINIA, Charleston WISCONSIN, Milwaukee INTERNATIONAL — EUROPEAN AREA HEADQUARTERS: Geneva, Tel: [41] (22)-93-33-11 INTERNATIONAL AREA HEADQUARTERS: Acton, MA 01754, U.S.A., Tel: (617) 263-6000 AUSTRALIA, Adelaide, Brisbane, Canberra, Hobart, Melbourne, Perth, Sydney, Townsville AUSTRIA, Vienna BELGIUM, Brussels BRAZIL, Rio de Janeiro, Sao Paulo CANADA, Calgary, Edmonton, Halifax, Kingston, London, Montreal, Ottawa, Quebec City, Regina, Toronto, Vancouver, Victoria, Winnipeg DENMARK, Copenhagen ENGLAND, Basingstoke, Birmingham, Bristol, Ealing, Epsom, Leeds, Leicester, London, Manchester, Reading, Welwyn FINLAND, Helsinki FRANCE, Bordeaux, Lyon, Paris, Puteaux, Strasbourg HOLLAND, Amstelveen, Delft, Utrecht HONG KONG IRELAND, Dublin ISRAEL, Tel Aviv ITALY, Milan, Rome, Turin JAPAN, Osaka, Tokyo MEXICO, Mexico City, Monterrey NEW ZEALAND, Auckland, Christchurch, Wellington NORTHERN IRELAND, Belfast NORWAY, Oslo, PUERTO RICO, San Juan SCOTLAND, Edinburgh REPUBLIC OF SINGAPORE SPAIN, Barcelona, Madrid SWEDEN, Gothenburg, Stockholm SWITZERLAND, Geneva, Zurich, WEST GERMANY, Berlin, Cologne, Frankfurt, Hamburg, Hannover, Munich, Nurnberg, Stuttgart