digital

# VAX11 780

## ARCHITECTURE HANDBOOK



digital VAX 11/780 ARCHITECTURE HANDBOOK VOL. 1 1977-78

PRINTED IN USA EB 07466 20/77 110 23 031

# VAX 11/780

# ARCHITECTURE HANDBOOK

digital

# CONTENTS

**CHAPTER 12 EXCEPTIONS**



**CHAPTER 13 PRIVILEGE INSTRUCTIONS**



**APPENDIX A DATA TABLES**

**VAX-11/780 SYSTEM**

# CHAPTER 1

# SYSTEM OVERVIEW

## 1.1 INTRODUCTION

The VAX-11/780 is the most powerful computer system in the -11 family of interactive computers, which includes the LSI-11, the PDP-11, and now the VAX-11. It consists of the VAX-11/780 processor and the VAX/VMS virtual memory operating system. It is designed for applications which require the power and sophistication of a high-performance, virtual memory computer system—at prices well below computer systems of its same caliber. It can be used as a powerful computational tool for high-speed, time-critical applications, for timesharing applications, and for a wide variety of commercial applications.

VAX, or Virtual Address eXtension, is the architecture for the VAX-11/780. It has been designed and developed by both hardware and software engineers and, in fact, was carefully documented before any implementation was begun. The goals of the VAX architecture were to provide a significant enhancement to the virtual addressing capability of the PDP-11 series consistent with small code size, easy exploitation by higher-level languages, and a high degree of compatibility with the PDP-11 series. While the VAX-11 is not strictly binary compatible with the PDP-11 binary code, it does implement a Compatibility Mode which executes most of the PDP-11 instructions (refer to Volumes 2 and 3). A consequence of this is that most user-level programs can execute in this Compatibility Mode, with system services and memory management being provided by the VAX/VMS operating system in Native Mode.

The VAX-11 architecture is characterized by a powerful and complete instruction set of 244 basic instructions, a wide range of data types, an elegant set of addressing modes, full demand paging memory management, and a very large virtual address space of over 4 billion bytes (2**32 bytes). Arithmetic and logical operations can be performed on byte-integers (a byte is eight bits), word-integers, and 32-bit longword-integers; plus, some instructions can perform operations on 64-bit quadword-integers. Additionally, the Native Mode instruction set includes floating point operations, character string manipulations, packed decimal arithmetic, and many instructions which improve the performance and memory utilization of systems and applications software. Some of these directly implement frequently used higher-level language constructs, such as DO loop control and the FORTRAN COMPUTED GO TO statement. There are also a number of operations which can be performed on variable-length bit fields, a new data type for the -11 family.

The other significant feature of the VAX-11 architecture is that unlike a very large class of computer systems, addressing for instructions is very nearly arbitrary. This means that there are no fixed formats—no restrictions as to the location of an operand for a particular instruction or even

the instruction itself. Thus, operands and instructions can begin on any byte address—odd or even. It is quite reasonable to express the location of any operand as being a register or a pair of registers in memory or by using indirection in memory. The result of this flexibility is that higher-level language compilers, such as FORTRAN, can generate code that is very small, very efficient, and easy to manipulate in the compiler's data structures. This means greater performance and lower memory utilization to accomplish the same task than on other computer systems in the same price class. These are but a few of the key features of the VAX-11/780's hardware architecture. The VAX/VMS operating system makes all the hardware work together as one unit to provide the VAX-11/780 with its multi-user, multiprogramming, virtual memory capabilities.

But before discussing the VAX-11/780's virtual memory capabilities more closely, a few definitions of some important terms will be valuable. One of the advantages of a virtual memory system is that an entire program does not have to be resident in main memory at one time. This means that portions of a program can be on the system disk and other portions in main memory. Programs are divided into small pieces called *pages*—512 bytes. A *process* is a collection of pages which runs a program; it consists of an address space plus both hardware and software context. That part of a process which is resident in main memory is called the process' *working set*. At any given point in time, there are many processes running on the system. The assemblage of processes which are resident in main memory is called the *balance set*. The action of bringing pages into and out of main memory is called *paging;* that of bringing complete working sets into and out of main memory is called *swapping*.

In order to control the simultaneous processing of many large programs, the VAX-11/780 incorporates sophisticated virtual memory management capabilities. VAX-11 memory management system is a tightly coupled hardware/software function. The hardware performs the task of translating from virtual addresses into physical ones. The VAX/VMS operating system provides the capabilities for paging, swapping, overlaying, protection, and sharing. Despite the fact that VAX/VMS performs these functions, the user can exert considerable control over the environment in which programs operate, i.e., the amount of paging and swapping that occurs during the execution of a program. Pages can be *locked*, or fixed, i.e., not candidates for removal, in a process' working set; pages can also be locked in main memory, and an entire working set can be locked in the balance set. Additionally, a user can specify that not one, but a number of pages be brought into main memory when a reference is made to a page which is not currently in main memory. This is called *clustering*. All of these tools allow a user to manipulate the environment in which a program executes to produce predictable performance—to provide fast, guaranteed response to external conditions.

Protection and sharing were key considerations in the development of the VAX-11 architecture. Both protection and sharing are at the page level. In a computer system, security and privacy are achieved by a combination of operations management and applications design. VAX/VMS complements this by providing the necessary system level reli-

ability and protection. Reliability is achieved by taking advantage of the hardware "firewalls." These "firewalls" include the four memory management access modes and the process structure.

One of the most important forces at work during the design and development of the VAX-11/780 system was an extensive reliability, availability, maintainability program (RAMP). This program affected all aspects of the product—the design of the basic hardware and software architectures right through to the end result—the VAX-11/780. Some of the significant RAMP features of the VAX-11 architecture and VAX/VMS are listed below. [Refer also to Chapter 2 and Volume 2 for details on the VAX-11/780 processor-specific RAMP features.]

1. Memory Management
   - 512 byte pages (protection, sharing, allocation)
   - four hierarchical access modes
   - read/write access control for each protection mode
   - access control violation produces a fault

2. Consistency and Error Checking
   - arithmetic traps for over- and underflow plus division by zero
   - limit checking traps to ensure the range referenced by certain instructions is valid
   - reserved operand trap to detect unacceptable data
   - interrupt return checks for detecting system malfunctions
   - string length checks

3. Special Instructions
   - Cyclic redundancy check (CRC) which provides a consistent method for performing software check-summing
   - CALL/RETurn provides a uniform standard for interfacing between local subroutines and system services

4. Maintenance Devices
   - high-resolution programmable real-time clock for scheduling and for diagnostics
   - processor identification register contains the processor's serial number and its latest hardware update status
   - time-of-year clock in conjunction with VAX/VMS enables unattended automatic restart in cases of power failure or fatal software error

5. Software
   - system software consistency checks detect and log operating system malfunctions and determine the validity of system control information
   - device interrupt timeout on all input/output avoids system hang if the interrupt is lost
   - disk bad block handling protects the integrity of data stored on disks
   - automatic retry on I/O errors
   - redundant recording of critical disk structures helps prevent the loss of the entire disk
   - error logging of hardware and software errors

- automatic restart capabilities
- on-line diagnostics for verification of peripherals while the operating system is performing other tasks

All of these capabilities—very large virtual address space, elegant and powerful instruction set, data types, and addressing modes, arbitrary byte addressing, full virtual memory paging with user control, integral protection and sharing, and extensive RAMP—have been combined in the VAX-11/780 to produce a product which can be applied to a wide range of demanding high-performance applications.

## 1.2 VAX-11/780 HANDBOOK SET

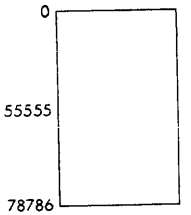
The VAX-11/780 handbooks comprise a three-volume set of detailed information on the system architecture, the VAX-11/780 system components, and the VAX-VMS virtual memory operating system. Each handbook concludes with an extensive glossary of terms commonly used in that handbook. This handbook, Volume 1, describes the entire hardware architecture needed by an assembly language programmer who writes non-privileged programs, i.e., those which do not directly use memory management or perform direct I/O. It provides information on the addressing modes, data representations, and the instruction set in sufficient depth to design and write applications and compilers. An overview of memory management, input/output programming, and an introduction to the VAX-11/780 processor components is provided.

Volume 2 provides documentation on VAX-11/780 hardware necessary to write privileged programs—details on memory management, process switching, input/output, processor registers, and compatibility mode. The chapters on input/output provide full programming details on the UNIBUS and MASSBUS adaptors. There is a chapter on the integral LSI-11 diagnostic console and the console command language. RAMP is covered in some detail and the peripherals supported by the VAX-11/780 system are described.

Volume 3 provides a uniform, cohesive description of the VAX-11 software—the VAX/VMS virtual memory operating system and its supported products. Primarily, it acts as an introduction to the VAX-11 software. It contains information on memory management, I/O file services, utilities and high-level languages, interprocess communication, process scheduling and context switching, command language, system services, plus interrupts, handlers, and asynchronous system traps.

## 1.3 NOTATIONAL CONVENTIONS

This section provides information on notational conventions used throughout the handbook set. Representations of memory, both physical and virtual, begin with low memory at the top of the diagram and progress toward higher addresses:




Unless otherwise noted, all numerical quantities are shown in decimal representation; decimal is the default radix of the system. Other representations are shown by 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 $\leftarrow$ sum + add + C

This shows the operation of adding the quantities "sum," "add," and "C" (for carry) and placing the result in "sum." Full details of this notation are given in Appendix E.

# CHAPTER 2

# VAX-11/780 INTRODUCTION

## 2.1 HARDWARE ARCHITECTURE

The VAX-11/780 computer system consists of the central processing unit (with integral floating point, packed decimal, and character string instructions), the console subsystem, the main memory subsystem, and the I/O subsystem. The I/O subsystem includes the Synchronous Backplane Interconnect (SBI)—an internal connection path which links the CPU with its subsystems.

These elements are illustrated in Figure 2-1.

## 2.2 THE SYNCHRONOUS BACKPLANE INTERCONNECT

As Figure 2-1 shows, all major hardware components are connected through the SBI, an internal synchronous path. This connection path, along with the VAX-11/780 central processor and the SBI devices (the adaptors and controllers shown in the figure), operates on clocked 200 nanosecond cycles. Thus, all transactions in the system are synchronized and occur at defined points in time.

The SBI is the primary control and data transfer path in the VAX-11/780 system. The SBI has a physical address space of 1 gigabyte (30 bits of address).

Physical address space is all possible memory and I/O addresses that a processor can access. In the VAX-11/780 system, half of the physical address space is for memory addresses and half for I/O addresses, as shown in Figure 2-2.

Of the 512 million bytes of memory which can be addressed, up to 2 million bytes may be connected to a VAX-11/780 system. I/O registers and memory can be addressed by instructions just as on the other PDP-11 family machines.

The SBI is capable of an aggregate data throughput rate of 13.3 million bytes/second and it cycles at 200 nanoseconds.

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

- *Distributed arbitration.* Since each device connected to the SBI determines whether or not it will receive the next cycle (rather than a

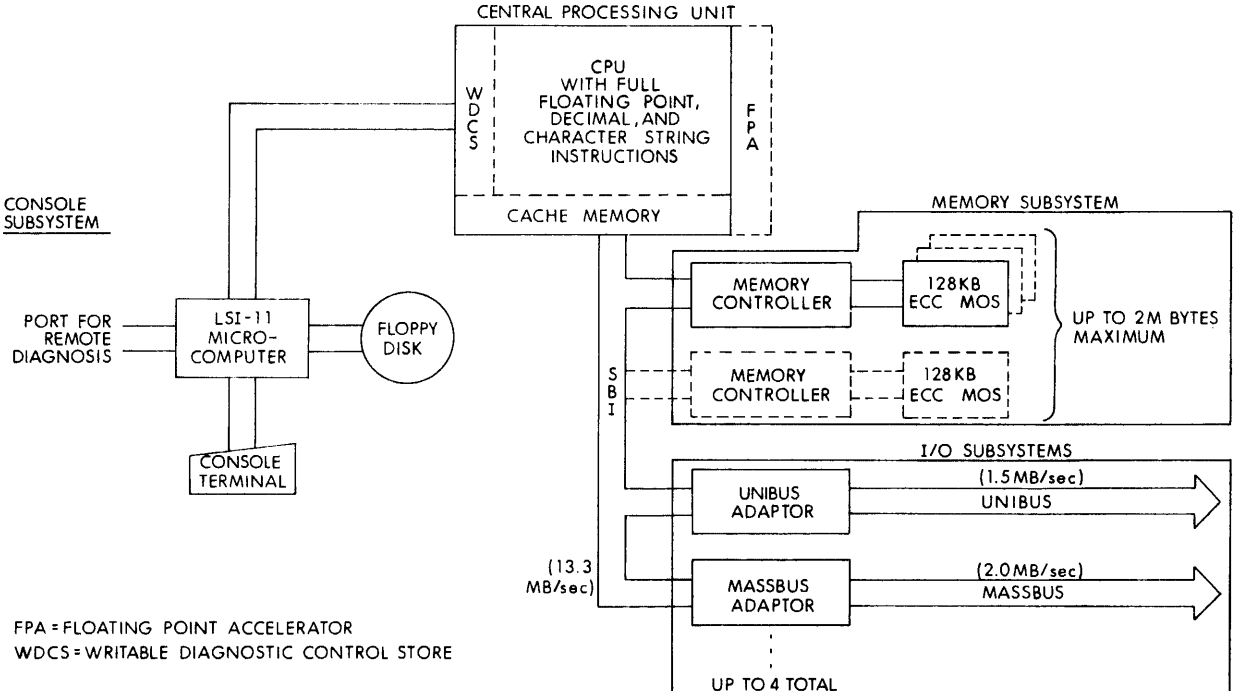



Figure 2-1 VAX-11/780 Hardware Architecture

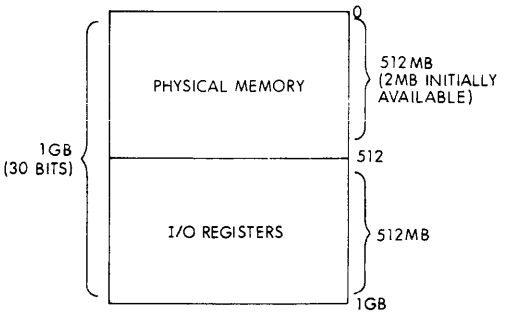



Figure 2-2 SBI Physical Address Space

central arbitrator making the decision), signals need travel the length of the SBI only once with the advantage of increased speed. Additionally, devices perform a parity check on the control information to assure that the arbitration is proceeding correctly.

- *Single 32-bit and two back-to-back 32-bit transfers.* The SBI data path is 32 bits wide. The protocol allows single (32-bit) and double (64-bit) data transfers as transactions. (The I/O adaptors always try data in 64-bit quadwords.)

Every transaction on the SBI (i.e., data transfer, address transfer, or command transfer) is parity checked and confirmed by the receiver. In addition, substantial protocol checking occurs on every cycle for high data integrity. This means the SBI preserves the integrity of the data it receives and transmits. (Data which is transferred from MASSBUS devices also includes parity; data from UNIBUS devices does not.)

Finally, a history of the last 16 SBI cycles is maintained by the CPU. This is an extremely useful aid in isolating system failures.

## 2.3 THE VAX-11/780 CENTRAL PROCESSING UNIT

The VAX-11/780 central processor is a high-speed, microprogrammed 32-bit computer that supports many of the features usually found only in larger systems (for example, support of many data types and virtual memory capabilities). The VAX-11/780 central processor executes VAX-11 variable length instructions in native mode, and non-privileged PDP-11 instructions in compatibility mode. The processor can directly address 4 gigabytes of virtual address space, and provides a complete and powerful instruction set that includes integral decimal, character string, and floating point instructions. The VAX-11/780 includes an 8K byte cache, integral memory management, sixteen 32-bit general registers, 32 interrupt priority levels, and an intelligent console (LSI-11).

Figure 2-3 illustrates the elements of the central processing unit.

## 2.4 NATIVE INSTRUCTION SET

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

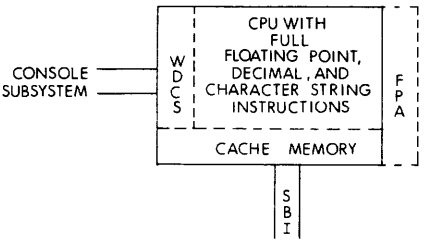



Figure 2-3 VAX-11/780 Central Processor

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

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

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

   Integer, floating point, packed decimal, and character data may be stored on an arbitrary byte boundary. Bit and bit-field data does not necessarily start on a byte boundary; this data type allows a collection of data structures to be packed together to use less storage space.

2. Instructions to manipulate special kinds of data—These include queue manipulation instructions (for example, those that insert and remove queue entries), address manipulation instructions, and user-programmed general register load and save instructions. These instructions are used extensively by the VAX/VMS operating system.
3. Instructions to provide basic program flow control—These include branch, jump, and case instructions, subroutine call instructions, and procedure call instructions.
4. Instructions to quickly perform special operating system functions—These include process control instructions (such as two special context switching instructions which allow process context variables to be

loaded and saved using only one instruction for each operation), and the Find First instruction which (among other uses) allows the operating system to locate the highest priority executable process. These instructions contribute to rapid and efficient rescheduling.

5. Instructions provided specifically for high-level language constructs—During the design of the VAX-11 architecture, special attention was given to implementing frequently used, higher-level language constructs as single VAX-11 instructions. These instructions contribute to decreased program size and increased execution speed. Some of the constructs which have become single instructions on the VAX-11/780 include:
   - the FORTRAN computed GOTO statement (translates into the CASE instruction)
   - the loop construct (for example, add, compare, and branch translates into the ACB instruction)
   - an extensive CALL facility (which aligns the stack on a longword boundary, saves user-specified registers, and cleans up the stack on return; the CALL facility is used compatibly among all native mode languages and operating system services).

VAX-11/780 instructions and data are variable length. They need not be aligned on longword boundaries in physical memory, but may begin at any byte address (odd or even). Thus, instructions that do not require argument use only one byte, while other instructions may take two, three, or up to 30 bytes depending on the number of arguments and their addressing modes. The advantage of byte alignment is that instruction streams and data structures can be stored in much less physical memory.

The VAX-11/780 processor offers nine addressing modes that use the general registers to identify the operand location:

register

register deferred

autoincrement

autoincrement deferred

autodecrement

displacement ..... (same as the PDP-11 index mode)

displacement deferred .... (same as the PDP-11 index deferred mode)

index (uses a second register scaled according to data type to provide true post-indexing capability)

literal (used for greater efficiency to specify small integer or floating point constants)

The hardware implements 8-, 16-, and 32-bit displacement for each of displacement and displacement deferred. This uses the minimal space for any memory reference. By combining modes, the programmer can achieve more addressing flexibility.

The instruction set is very consistent and the assembler mnemonics are

clear. Programmers who are already familiar with the PDP-11 instruction set will find the VAX-11 instruction formats similar, as well as the data formats and the use of addressing modes, general purpose registers, and stacks. Thus, the amount of programmer retraining that is required is minimized. Those programmers who are not familiar with the PDP 11 programming style should find that the consistency and power of the VAX-11 instruction set allows them to be producing efficient executable code quickly.

Because the instruction set is so flexible, fewer instructions are required to perform any given function. The result is more compact and efficient programs, faster program execution, faster context switching, more precise and faster math functions, and improved compiler-generated code.

## 2.5 COMPATIBILITY MODE INSTRUCTION SET

In addition to its 32-bit native mode instruction set, the VAX-11/780 processor can concurrently execute a subset of the PDP-11 instruction set in compatibility mode. This is not done by emulation or simulation; both instruction sets are built into the microcode and logic of the processor.

The PDP-11 instruction set implementation is a subset of the PDP-11/70's. Specifically, it contains all instructions except those which perform the following functions:

1. Execution of floating-point instructions.
2. Use of both instruction (I) space and data (D) space.
3. Execution of privileged functions such as:
   - HALT, RESET and special instructions, such as traps and WAIT, which are normally reserved for operating system usage
   - Direct access to internal processor registers such as the Processor Status Word and the Console Switch Register
   - Direct access to the trap and interrupt vectors which must be initialized for interrupt servicing
   - Execution in any mode other than User mode along with the corresponding access to the alternate general register set

## 2.6 GENERAL REGISTERS AND STACKS

The VAX-11/780 CPU provides sixteen 32-bit general registers which can be used for temporary storage, as accumulators, index registers, and base registers. Although all can be used as general-purpose registers, four have special significance depending on the instruction being executed: Register 12 (the CALL argument pointer); Register 13 (the CALL frame pointer); Register 14 (the stack pointer); and Register 15 (the program counter).

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

## 2.7 CACHES

The VAX-11/780 CPU provides three "cache" systems—the memory cache, an address translation buffer, and an instruction buffer.

### 2.7.1 Memory Cache

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

The memory cache stores 8K bytes and is implemented as a two-way set-associative write-through cache. This cache also watches I/O transfers on the SBI and updates itself appropriately. Thus, no operating system overhead is needed to synchronize the cache with I/O operations, since the cache resolves all these stale data problems.

For reliability reasons the VAX-11/780's memory cache uses the write-through technique for updating main memory. With this method, whenever a write reference occurs, the data is not only stored in the cache itself, but is also immediately copied into the backing store (main memory). This means that main memory always contains a valid copy of all data in the cache. Normally this would mean that the CPU would have to suspend processing until main memory has accepted the write data. In the VAX-11/780, however, the central processor's interface to the SBI includes a 32-bit write buffer. Therefore, when a write reference occurs, the CPU stores the write data in the buffer, initiates a write transfer to main memory, and continues with the next instruction.

### 2.7.2 Instruction Buffer

The instruction buffer consists of an 8-byte buffer that enables the CPU to fetch and decode the next instruction while the current instruction completes execution. The instruction buffer in combination with the parallel data paths (which can perform integer arithmetic, floating point operations, and shifting all at the same time) significantly enhances the VAX-11/780's performance.

### 2.7.3 Translation Buffer

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

**Standard Schottky TTL Logic** The VAX-11/780 system uses Schottky TTL logic circuits, proven technology that combines fast switching speed with moderate power consumption. Emitter-coupled logic circuits and custom large-scale integrated circuits have been used where appropriate to optimize system performance and reliability.

**Clocks** The standard VAX-11/780 CPU includes two clocks—a high-precision, programmable real-time clock used by system diagnostics and

by the VAX/VMS operating system for accounting and scheduling, and a time-of-year clock, which insures the correct time of day and date. The time-of-year clock additionally includes a battery which provides backup for over 150 hours. The time-of-year clock is used by the operating system to enable unattended automatic restart following any service interruption, including a power failure.

**Writable Diagnostic Control Store (WDCS)** 12K bytes (plus parity) of WDCS are provided to allow the Diagnostic Console Microcomputer to verify the integrity of crucial parts of the system (for example, the key parts of the CPU, the intelligent console, the SBI, and the memory controller). In addition, the WDCS can be used to implement updates to the VAX-11/780's microcode. In this way, DIGITAL can keep customers up-to-date with corrections.

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

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

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

## 2.8 HIGH PERFORMANCE FLOATING POINT ACCELERATOR

The VAX-11/780 floating point accelerator option operates in parallel with the CPU and transparently to programs. It executes the standard floating point instruction set, add, subtract, multiply, and divide, in both single- and double-precision formats, plus three additional instructions. These are extended multiply and integerize (EMOD), polyonomial evaluation (POLY), single- and double-precision formats for both instructions, and 32-bit integer multiply (MULL). EMOD is used for fast, accurate range reduction of mathematical function arguments. POLY is used extensively by the math library in the evaluation of such mathematical functions as sine, cosine, etc. Subscript calculations can be done fast and efficiently using the MULL instruction.

An additional 12K bytes (or 1,024 microwords = 96 data bits plus three parity bits) of writable control store is available for customer applications. There are, however, no software tools or supporting documentation for this option.

## 2.9 THE MEMORY SUBSYSTEM

The main memory subsystem consists of ECC MOS memory, which is connected to the SBI via the memory controller, as illustrated in Figure 2-4.

MOS memory may be added in increments of 128K byte units to a maximum of 1 million bytes per controller. Two memory controllers may be

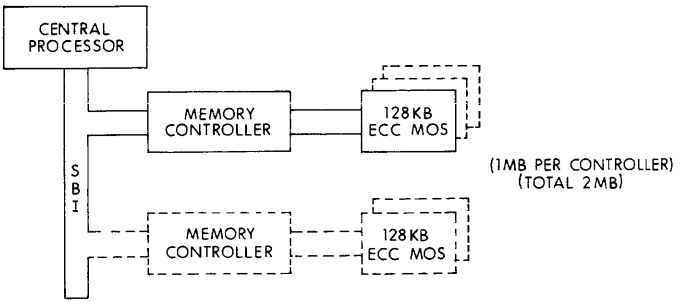



Figure 2-4 VAX-11/780 Memory Subsystem

connected to a VAX-11/780 system, for a total of 2 million bytes of physical memory. (The minimum memory requirement is 128K bytes.)

The VAX-11/780 physical memory is built using 4K MOS RAM chips. It is organized in quadwords (64 bits) plus an 8-bit ECC (Error Correcting Code), which allows the correction of all single-bit errors, and the detection of all double-bit errors and approximately 70% of greater than double-bit errors, providing a ten-fold improvement in MTBF.

The memory cycle time is 600 nanoseconds. This is equal to the memory access time, since MOS memory has non-destructive read-out. Read access time at the central processor (including SBI overhead) is 1800 nanoseconds. This is measured from the time the processor transmits a read request until the processor receives all 64 bits of data. (The central processor always reads 64 bits from memory.) In spite of the 1800 nanosecond memory access time, the VAX-11/780 processor realizes an effective average operand access time of 290 nanoseconds, because of its large optimized memory cache.

The memory controllers allow the writing of data in full 32- and 64-bit units. Also, upon command from an SBI device, individual bytes (or a single byte) may be written. Each memory controller buffers up to four memory access requests. This "request buffer" substantially increases memory throughput and overall system throughput and decreases the need for interleaving for most configurations. With this buffer, memory bandwidth essentially matches that of the SBI—13.33 million bytes/second, including time for refresh cycles. This is because a number of transactions may occur concurrently. For example, the memory controller may accept a WRITE command from a MASSBUS adaptor while it is reading previously requested data by the processor for increased throughput. Were it not for the request buffer, there would be about a 50% degradation in memory bandwidth, making interleaving necessary to approach the SBI bandwidth.

Interleaving is possible with two controllers and equal amounts of memory on each. Interleaving is enabled/disabled under program control. It is performed at the quadword level (each 64 bits) because of the memory organization.

The integrity of data in the VAX-11/780's ECC MOS memory is retained upon power interruption in two ways. Firstly, the memory modules are connected to an unswitched power supply so that when the system is turned off, refreshing is continued.

Secondly, in the case of a temporary power failure, the contents of MOS memory may be protected using optional battery backup. Each DIGITAL-supplied option preserves 1 million bytes of memory for a maximum of ten minutes. If the system has less than 1 million bytes of memory, the battery supplies longer backup time. In addition, customer-supplied battery backup may be used with the DIGITAL option to prolong backup time.

## 2.10 THE INPUT/OUTPUT SUBSYSTEMS

The VAX-11/780's I/O subsystems consist of the UNIBUS and MASSBUS and their respective adaptors through which I/O devices communicate. As shown in Figure 2-5, each VAX-11/780 system has one UNIBUS adaptor and can have up to four MASSBUS adaptors.

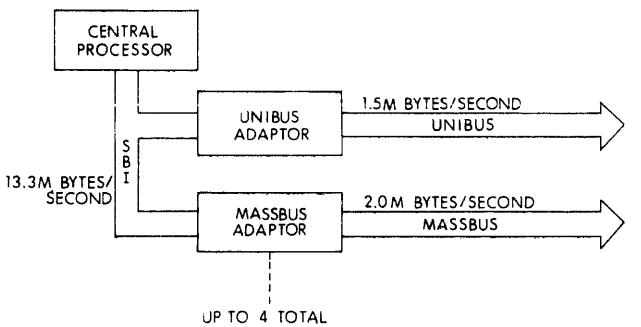



Figure 2-5 VAX-11/780 I/O Subsystem

### 2.10.1 The UNIBUS

General-purpose and customer-developed devices are connected to the VAX-11/780 system via the VAX-11/780's UNIBUS. Since the SBI deals in 30-bit addresses (1 gigabyte), 18-bit UNIBUS addresses must be translated to 30-bit SBI addresses. This mapping function is performed by the UNIBUS adaptor, a special interface between the SBI and the UNIBUS, which translates UNIBUS addresses, data, and interrupt requests to their SBI equivalents, and vice versa.

The UNIBUS adaptor does priority arbitration among devices on the UNIBUS, a function handled by logic in the PDP-11 CPUs. The address translation map permits contiguous disk transfers to and from noncon-

tiguous pages of physical memory (these are called scatter/gather operations).

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

Any number of unbuffered DMA transfers are handled by one direct DMA data path. Every 8- or 16-bit transfer on the UNIBUS requires a 32-bit transfer on the SBI (although only 16 bits are used). The maximum transfer rate through the Direct Data Path is 750 thousand bytes/second.

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

### 2.10.2 The MASSBUS(es)

High-performance mass storage devices, such as the RP series and RM moving head disks, are connected to the VAX-11/780 system using a MASSBUS adaptor. The MASSBUS adaptor is the interface between the MASSBUS and the SBI and performs all control, arbitration, and buffering functions. Address mapping is similar to that performed by the UNIBUS adaptor.

There may be a total of four MASSBUS adaptors on each VAX-11/780 system. Each adaptor can accommodate data transfers of 128K bytes maximum to and from noncontiguous pages in physical memory (scatter/gather). The VAX/VMS operating system supports transfers of 65KB maximum to be consistent with other devices.

Each MASSBUS adaptor uses a 32-byte silo data buffer, which permits transfers at rates up to 2 million bytes/second to and from physical memory (8MB/second with all four). As in the UNIBUS adaptor, data is assembled in 64-bit quadwords (plus byte parity) to make maximum efficient use of the SBI bandwidth.

On memory-to-MASSBUS transfers, as on memory-to-UNIBUS transfers, the adaptor anticipates upcoming MASSBUS data transfers by pre-fetching the next 64 bits of data from memory.

The combination of UNIBUS and MASSBUS transfer rates gives a maximum throughput of 9.5 million bytes/second to and from the SBI. Thus, there is ample bandwidth remaining (3.8 million bytes/second) to handle the central processing unit (which typically uses 1 million bytes/second).

## 2.11 THE CONSOLE SUBSYSTEM

The VAX-11/780's integral console consists of an LSI-11 microcomputer with 16K bytes of read/write memory and 8K bytes of ROM (used to store the LSI diagnostic, the LSI bootstrap, and fundamental console routines), a floppy disk (for the storage of basic diagnostic programs and for software updates), a terminal, and a 20mA serial line interface (for the console terminal). Remote access by a DIGITAL Diagnostic Center is available.

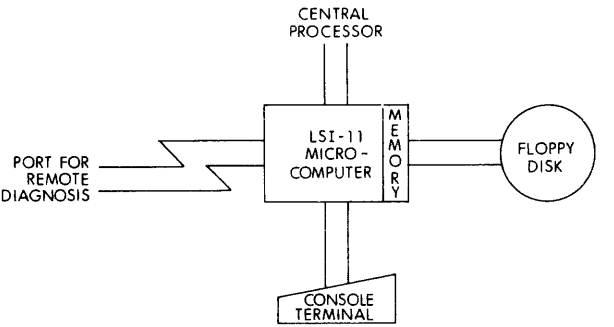



Figure 2-6 VAX-11/780 Console Subsystem

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

A floppy disk is included with every VAX-11/780 system. It is used for a variety of functions:

- During system installation, it acts as a load device. The LSI-11 ROM bootstrap reads a file from the floppy which, in turn, is used to load the operating system.
- It stores the system hard-core diagnostics, namely the hardware verification programs for the LSI-11 itself, CPU, SBI, a memory controller, and a memory module.

  These diagnostics are run upon command at power-up time to verify the integrity of the system hard-core. On-line diagnostics, which run under the VAX/VMS operating system, are then run to verify other system components.
- The floppy is also used to distribute updates, or modifications, to the system software. The updates are provided in machine-readable form

so that with a few simple commands from the system console, the information on the floppy can be automatically read in and used by VAX/VMS to update itself.

## 2.12 RELIABILITY-AVAILABILITY-MAINTAINABILITY PROGRAM (RAMP)

Major consideration was given to product quality (i.e., reliability, usability, serviceability, etc.) throughout the planning and development stages of the VAX-11/780 project. The system designers early adopted a policy of quality "insurance" (build the right features into the product) in addition to quality "assurance" (check to see that they are there). In particular, their goal was to build a product that is:

- extremely reliable
- highly available (i.e., with minimal down-time)
- with improved hardware and software warranty/maintenance procedures

Their method was to design a system with better, more complete, and easier-to-use diagnostics, documentation, system safeguards, and maintenance procedures than currently exist in any minicomputer competitive product.

VAX-11/780 RAMP features are summarized below under four major categories: Hardware Architecture, Improved Packaging, Diagnostic Aids, and Software Architecture. For greater detail refer to Volume 2.

### 2.12.1 Hardware Architecture

- Four Hierarchical Access Modes (kernel, executive, supervisor, and user) protect system information and improve system reliability and integrity.
- A Diagnostic Console, consisting of an LSI-11 microcomputer, floppy diskette, and console terminal, provides both local and remote diagnosis of system errors and simplifies system bootstrap and software updates. Simple console commands replace lights and switches. The diagnostic console provides faster and easier maintenance procedures and increases availability.
- Automatic Consistency and Error Checking detects abnormal instruction uses and illegal arithmetic conditions (overflow, underflow, and divide by zero). Continual checking by the hardware (and uniform exception handling by the software) increases data reliability.
- Special Instructions, such as CALL and RETURN, provide a standard program calling interface for increased reliability.
- Integral Fault Detection and Maintenance Features, including:

  ECC on memory corrects all single-bit errors and detects all double-bit errors to increase availability and aid in maintenance.

  ECC on the RM03, RP05, RP06, and RK06 disks detects all errors up to 11 bits and corrects errors in a single error burst of 11 bits.

  An SBI history silo maintains a history of the sixteen most recent cycles of bus activity and may be examined to aid in probem isolation.

Maintenance registers permit forced error conditions for diagnostic purposes.

A high resolution programmable real-time clock permits testing of time-dependent functions.

Extensive parity checking is performed on the SBI, MASSBUS, and UNIBUS adaptors, memory cache, address translation buffer, microcode, writable diagnostic control store, and key CPU buses and registers.

A watchdog timer in the LSI-11 diagnostic console detects hung machine conditions and allows crash/restart recovery actions.

Clock margining provides diagnostic variation of the clock rate and aids in problem isolation.

Disabling of the memory management and the cache aids in isolating hardware problems.

- Fault Tolerance Features, including:

  Detection and recording of bad blocks on disk surfaces to increase the reliability of the medium.

  Write-verify checking hardware in peripherals available to verify all input and output disk and tape operations and to ensure data reliability. Track offset retry hardware to enable programmed software recovery from disk transfer errors.

### 2.12.2 Improved Packaging

- The VAX-11/780 System meets Underwriters Laboratory (U.S.A.), Canadian Standards Association and IEC requirements for data processing equipment. It has been designed for easy access and serviceability.
- Improved Air Flow increases system reliability while permitting easier on-line access to components needing maintenance. Servicing will not cause cooling problems.
- Power Loss, Temperature, and Air Flow Sensors detect emergency conditions and protect the system from damage. Indicators aid in diagnosis and maintenance.
- Subassembly Replacement of the power supply, logic subassembly, or blowers can be done by one person with common tools in less than 20 minutes.
- Cabling is located away from modules and fixed in cable troughs for greater protection from damage and less interference with cooling.
- A Modular Power Supply, with malfunction indicator lights, provides easier problem isolation.

### 2.12.3 Improved Diagnostic Aids

- Optional Remote Diagnosis capabilities (performed with the customer's permission) allow the field service engineer to examine the error log file, and load, run, and control all level diagnostics from a remote terminal, for reduced maintenance time and costs.
- System Verification Test Packages test device interactions and system integrity.

- Functional and Fault Isolation Diagnostics perform tests (upon request) of the "crucial" parts of the hardware, run device diagnostics, and verify the reliability of the hardware, to aid preventative maintenance and repair procedures.

### 2.12.4 Software Architecture

- Operating System Consistency Checks detect and log system malfunctions and determine the validity of system control information for increased system reliability.
- Redundant Recording of Critical Information (i.e., the home block and index file header) for increased volume reliability.
- Uniform Exception Handling, performed for both hardware and software exceptions, improves system reliability.
- On-Line Error Logging monitors hardware and software and notes error occurrences in a log file which can be examined and used as a maintenance aid.
- Unattended Automatic Restart Capabilities increase availability by bringing the system up automatically following a system crash or a power failure (operator can override).
- On-Line Software Update and Maintenance operations can be performed concurrently with other system activities for increased system availability.

# CHAPTER 3

# ARCHITECTURE

## 3.1 INTRODUCTION

This chapter describes the application programming environment, specifically that seen by the assembly language programmer. It is intended to introduce the programmer to those features of the VAX-11 architecture which directly affect the design of VAX-11 programs.

The VAX-11 architecture is intended to support multiprogramming, which is the concurrent execution of a number of processes in a single computer system. A process, loosely defined, is a single stream of machine instructions executed in sequence.

The virtual address space (that is, the memory space as it appears to a process) is mapped onto the physical address space (that is, the memory space which actually exists in the hardware) by the memory management logic in the processor. This logic also supports paging, by which the system keeps in physical memory only those parts of a process' virtual memory actively in use.

A VAX-11 process exists in and operates on a memory space of 2**32 (about 4.3 giga) bytes. Some addresses and data are kept in sixteen 32-bit general registers. A small number of processor state variables are kept in a special register called the Processor Status Longword, or PSL. This set of information (memory, general registers, and PSL) defines a process. This chapter will cover each in some detail, while subsequent chapters will describe the instructions and data which make up a VAX-11 process.

## 3.2 MEMORY

The memory space addressable by any program is 2**32 bytes (that is, virtual addresses are 32 bits long). Of that space, one half (that with the most significant bit set) is referred to as system space, because it is the same for all processes in the system. It is used for the operating system software and system-wide data. System space is shared by all processes to facilitate interrupt handling and system service routines.

The other half of the virtual address space (that with the most significant address bit clear) is separately defined for each process; it is therefore referred to as 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' working memory. Except for special cases of sharing, each process has its own P0 and P1 spaces, independent of others in the system. Figure 3-1 illustrates the address spaces of several processes in a multiprogramming system. Each pro-

cess space is independent of the others, while the system space is shared by all.

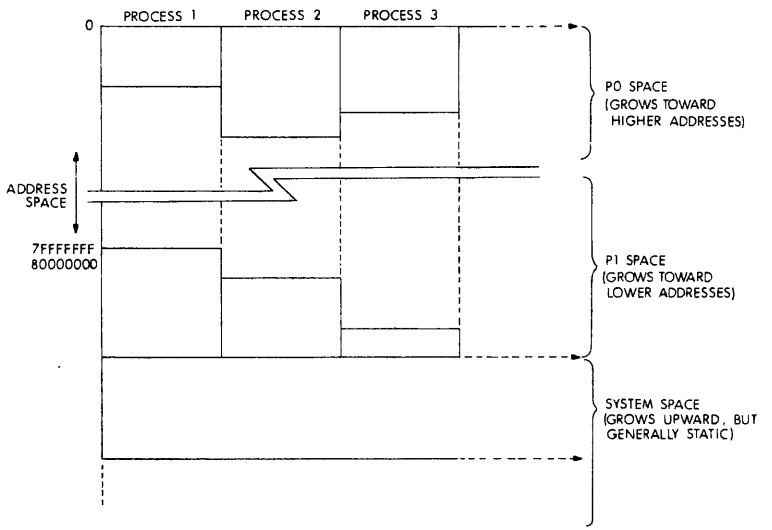



Figure 3-1 Address Spaces in Process Context

The basic addressable unit in VAX-11 is the 8-bit byte. Larger units are built up by doubling: a word is two bytes; a longword is four bytes; a quadword is eight bytes. These four sizes are the units in which VAX-11 memory stores data, even though the processor sometimes interprets operands in other units, such as half bytes, or nibbles, for decimal digits, or variable-sized bit fields.

In general, the memory system processes only requests 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 4, and a quadword can only come from an address which is a multiple of eight. All VAX-11 processors have a provision for converting an unaligned request into a sequence of requests that can be accepted by the memory. Users, however, should be aware that this conversion has a serious impact on performance, and should design their data structures in such a way that the natural alignment of operands is preserved wherever possible.

The VAX-11 memory management logic serves the following purposes:

- It allows a number of processes to occupy main memory simultaneously, all freely using process space addresses, but referring independently to their own programs and data.
- It allows the operating system to keep selected parts of a process and its data in memory, bringing in other parts as needed, without ex-

plicit intervention by the program. Large programs can be run in reduced memory space without recoding or overlays visible to the programmer.

- It allows the operating system to 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.
- It allows cooperating processes to 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.
- It allows the operating system to limit 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 less privileged routines cannot.
- It provides the means for the operating system to 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 space. Pages are always naturally aligned (that is, the address of the first byte of a page is a multiple of 512).


Figure 3-2: Virtual Address Format

Virtual addresses are 32 bits long, and are divided up by the memory management logic as shown in Figure 3-2. 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 Page Table Entry (PTE) from the P0, P1, or system page table as appropriate. The page table entry contains the following 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 (that is, that 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

In concept, the process of obtaining a page table entry occurs on every memory reference. In practice, however, the processor maintains a translation buffer. The translation buffer is a special-purpose cache of recently used page table entries. Most of the time, the translation buffer already contains the page table entries for the virtual addresses used by the program, and the processor does not need to go to memory to obtain the PTE (Page Table Entry).

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 page table entries for addresses greater than 80000000 (hex), 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, however, 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-11 defines the process space page tables, P0PT and P1PT, to be allocated in contiguous areas of system space—that is, 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. However, 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.

## 3.3 GENERAL REGISTERS

VAX-11 provides sixteen general registers for temporary address and data storage. Registers are denoted Rn, where n is a decimal number in the range 0 through 15. Registers do not have memory addresses, but are accessed either explicitly by inclusion of the register number n in an operand specifier, or implicitly by machine operations which make reference to specific registers. Certain registers have specific uses, and have special names always used by software:

PC — R15 is the Program Counter (PC). The processor updates it to address the next byte of the program; PC is therefore 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-11 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 containing the address of a stack frame. Further, VAX-11 software depends on maintenance of FP for correct reporting of certain exceptional conditions.

AP — R12 is the Argument Pointer (AP). The VAX-11 procedure call convention uses a data structure called an argument list, and uses 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 R6 through R11 have no special significance either to hardware or the operating system. Specific software will assign specific uses for each register.

R0-R5 — Registers R0 through R5 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, RC, and POLY instructions. The specific instruction descriptions identify which registers are used, and what values are loaded into them.

The general philosophy 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-11 procedure call convention regards R0 and R1 as so temporary that they are not even saved on calls.

## 3.4 STACKS

Stacks, also called pushdown lists or last-in-first-out queues, are an important feature of DIGITAL's -11 family 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-11 by a block of memory and a general register which addresses the "top" of the stack—that is, that location in

**Table 3-1 Special Register Usage**

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

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-11 instructions; thus any instruction can operate on the stack, and it is seldom necessary to devote separate instructions to maintenance of the stack pointer. See Chapter 5 for details of the addressing modes of VAX-11 instructions.

A stack is usually bounded by inaccessable 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.

Many VAX-11 processor operations make use of the stack implicitly (that is, 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 stack as is used 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 3-3 illustrates the relationships of the five stacks and multiple processes.

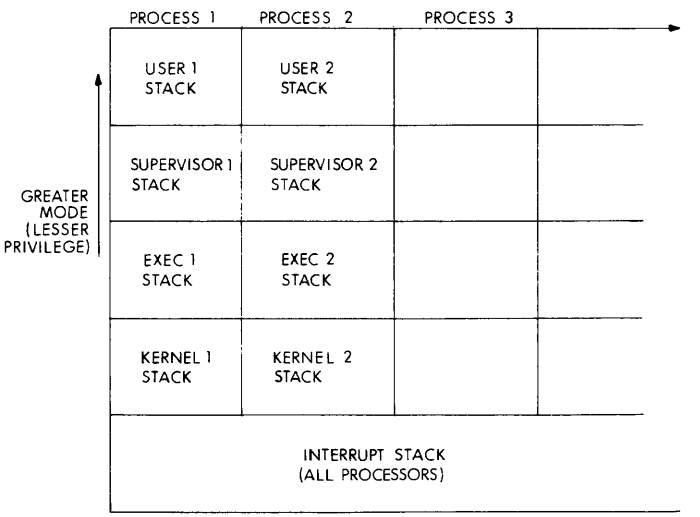



Figure 3-3 Stacks by Mode vs. Processes

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

User mode programs are not subject to sudden and non-reproduceable 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.

## 3.5 PROCESSOR STATUS LONGWORD

There are a number of processor state variables associated with each process, which VAX-11 groups 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 contain 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.

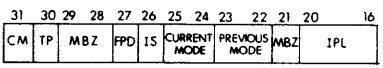



Figure 3-4 Processor Status Longword

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. Refer to the individual instruction descriptions in chapters 6 to 11 for details of how each instruction affects the condition codes. The condition codes are tested by the conditional branch instructions.

**N**—Bit 3 is the Negative 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 2 is the Zero condition code; in general 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 1 is the oVerflow 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 0 is the Carry condition code; in general 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 4-7 of the PSL are trap-enable flags, which cause traps to occur under special circumstances:

**T**—Bit 4 is the Trace 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.

**IV**—Bit 5 is the Integer oVerflow trap enable; when set, it causes an integer overflow trap after any 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).

**FU**—Bit 6 is the Floating Underflow 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.

**DV**—Bit 7 is the Decimal oVerflow 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**

There are other trap conditions for which there are no enable flags—division by zero and floating overflow.

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

**IPL**—Bits 16-20 represent the processor's Interrupt Priority Level. 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 at 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. Refer to Volume 2 for full details. Briefly, there are 31 priority levels above zero, numbered in hex 01 through 1F. Inter-

rupt levels 01 through 0F exist entirely for use by software. Levels 10 through 17 are for use by peripheral devices and their controllers, though present systems support only 14 through 17. Levels 18 to 1F are for use for urgent conditions, including the interval clock, serious errors, and power fail.

**Previous Mode**—Bits 22-23 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 only 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 24-25 are the current mode field, which determines the privilege level of the currently executing program. The values of mode are:

0—Kernel; most privileged, including the ability to perform all instructions
1—Executive
2—Supervisor
3—User; least privileged

Privileged is granted in two ways by the mode field—certain instructions (HALT, Move To Processor Register, and Move From Processor Register) and 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 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 mode. When IS is set, the current mode is always kernel; thus software operating "on the interrupt stack" has full kernel-mode privileges.

**FPD**—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 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. See Chapter 12 for a full discussion of TP.

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

# CHAPTER 4
# DATA REPRESENTATION

The VAX-11 instruction set deals directly with several data types. These can be separated into the integer, floating point, variable length bit field, character string, and decimal string classes. Most of these types can be subdivided into data types of differing sizes and formats.

The integer data types are used to represent in a binary format quantities that have a fixed scaling. These quantities can be treated as either signed or unsigned. When treated as signed quantities, integers are represented in twos complement form. This means that a negative number is one greater than the bit-by-bit complement of its positive counterpart. When treated as unsigned quantities, integers range from 0 through 2**n where there are n bits in the representation. VAX-11 supports in the instruction set integer data types of 8, 16, 32, and 64 bit sizes. These are termed byte, word, longword, and quadword integers respectively.

The floating point data types are used to represent approximations to quantities for which the scaling is not specified in the program. Floating point data is stored in a scientific notation as a power of two times a fraction in the range .5 (inclusive) to 1.0 (exclusive). The data representation consists of three fields, the sign, the power of two exponent, and the fractional magnitude. VAX-11 supports in the instruction set floating point data types of 32 and 64 bit sizes. These are termed floating and double floating respectively.

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.

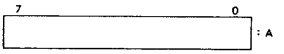
The character string is a data type used to represent strings of characters such as names, data records, or text. Rather than performing arithmetic or logical operations on character strings, the important operations include copying, concatenating, searching, and translating the string.

The decimal string data types are used to represent fixed scaled quantities in a form close to their external representation. For programs that are input/output intensive rather than computation intensive, this representation is frequently more efficient. The decimal string data types include formats in which each decimal digit occupies one byte (character) and a more compact form in which two decimal digits are packed into one byte. These are termed numeric and packed decimal strings respectively. Because the numeric string form represents many external

data arrangements exactly, it appears in several representations. The most significant distinguishing characteristic is whether the sign, if any, appears before the first digit or whether it is superimposed on the final digit. These are termed leading separate and trailing numeric strings respectively.

### 4.1 BYTE

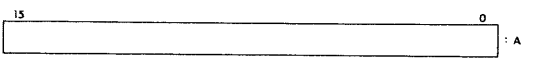
A byte is 8 contiguous bits starting on an addressable byte boundary. The bits are numbered from the right 0 through 7:




A byte is specified by its address A. When interpreted arithmetically, a byte is a twos complement integer with bits of increasing significance going 0 through 6 and bit 7 the sign bit. The value of the integer is in the range —128 through 127. For the purposes of addition, subtraction, and comparison, VAX-11 instructions also provide direct support for the interpretation of a byte as an unsigned integer with bits of increasing significance going 0 through 7. The value of the unsigned integer is in the range 0 through 255.

### 4.2 WORD

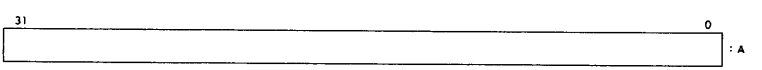
A word is 2 contiguous bytes starting on an arbitrary byte boundary. The bits are numbered from the right 0 through 15:




A word is specified by its address A, the address of the byte containing bit 0. When interpreted arithmetically, a word is a twos complement integer with bits of increasing significance going 0 through 14 and bit 15 the sign bit. The value of the integer is in the range —32,768 through 32,767. For the purposes of addition, subtraction and comparison, VAX-11 instructions also provide direct support for the interpretation of a word as an unsigned integer with bits of increasing significance going 0 through 15. The value of the unsigned integer is in the range 0 through 65,535.

### 4.3 LONGWORD

A longword is 4 contiguous bytes starting on an arbitrary byte boundary. The bits are numbered from the right 0 through 31:



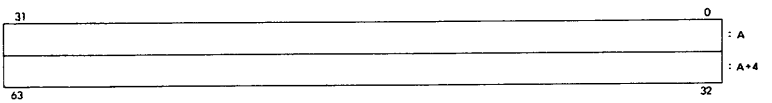

A longword is specified by its address A, the address of the byte containing bit 0. When interpreted arithmetically, a longword is a twos complement integer with bits of increasing significance going 0 through 30 and bit 31 the sign bit. The value of the integer is in the range

—2,147,483,648 through 2,147,483,647. For the purposes of addition, subtraction, and comparison, VAX-11 instructions also provide direct support for the interpretation of a longword as an unsigned integer with bits of increasing significance going 0 through 31. The value of the unsigned integer is in the range 0 through 4,294,967,295.

Note that the longword format is different from the longword format defined by the PDP-11 FP-11. In that format, bits of increasing significance go from 16 through 31 and 0 through 14. Bit 15 is the sign bit. Most DIGITAL software and in particular PDP-11 FORTRAN uses the VAX-11 longword format.

### 4.4 QUADWORD

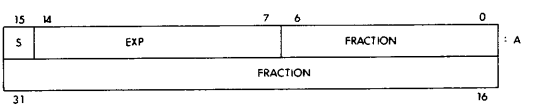
A quadword is 8 contiguous bytes starting on an arbitrary byte boundary. The bits are numbered from the right 0 through 63:




A quadword is specified by its address A, the address of the byte containing bit 0. When interpreted arithmetically, a quadword is a twos complement integer with bits of increasing significance going 0 through 62 and bit 63 the sign bit. The value of the integer is in the range —2**63 to 2**63-1. The quadword data type is not fully supported by VAX-11 instructions.

### 4.5 FLOATING

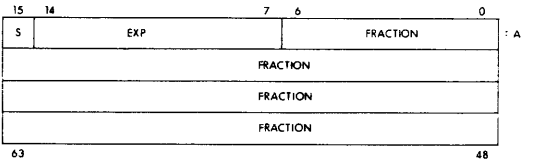
A floating datum is 4 contiguous bytes starting on an arbitrary byte boundary. The bits are labelled from the right 0 through 31.




A floating datum is specified by its address A, the address of the byte containing bit 0. The form of a 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 of increasing significance go 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, is taken to indicate that the floating datum has a value of 0. Exponent values of 1 through 255 indicate 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 (See Chapters 6 and 12). The value of a floating datum is in the approximate range .29*10**—38 through 1.7*10**38. The precision of a floating datum is approximately one part in 2**23, i.e., typically 7 decimal digits.

## 4.6 DOUBLE FLOATING

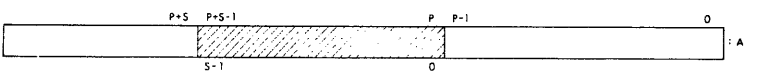
A double floating datum is 8 contiguous bytes starting on an arbitrary byte boundary. The bits are labelled from the right 0 through 63:



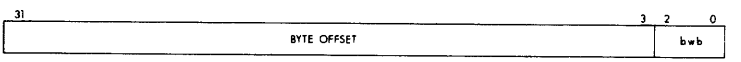

A double floating datum is specified by its address A, the address of the byte containing bit 0. The form of a double floating datum is identical to a floating datum except for an additional 32 low significance fraction bits. Within the fraction, bits of increasing significance go 48 through 63, 32 through 47, 16 through 31, and 0 through 6. The exponent conventions, and approximate range of values is the same for double floating as floating. The precision of a double floating datum is approximately one part in 2**55, i.e., typically 16 decimal digits.

## 4.7 VARIABLE LENGTH BIT FIELD

A variable bit field is 0 to 32 contiguous bits located arbitrarily with respect to byte boundaries. A variable bit field is specified by three attributes: the address A of a byte, a bit position P that is the starting location of the field with respect to bit 0 of the byte at A, and a size S of the field. The specification of a bit field is indicated by the following where the field is the shaded area.




The position is in the range −2**31 through 2**31−1 and is conveniently viewed as a signed 29-bit byte offset and a 3-bit bit-within-byte field:

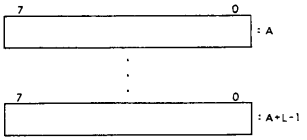



The sign extended 29-bit byte offset is added to the address A and the resulting address specifies the byte in which the field begins. The 3-bit bit-within-byte field encodes the starting position (0 through 7) of the field within that byte. The VAX-11 field instructions provide direct support for the interpretation of a field as a signed or unsigned integer. When interpreted as a signed integer, it is twos complement with bits of increasing significance going 0 through S−2; bit S−1 is the sign bit. When interpreted as an unsigned integer, bits of increasing significance go from 0 to S−1. A field of size 0 has a value identically equal to 0; it contains no bits and no memory is referenced; hence, the address need not be valid.

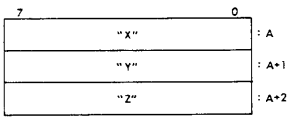
A variable bit field may be contained in zero to five bytes. From a memory management point of view only the minimum number of bytes necessary to contain the field is actually referenced.

### 4.8 CHARACTER STRING

A character string is a contiguous sequence of bytes in memory. A character string is specified by two attributes: the address A of the first byte of the string, and the length L of the string in bytes. Thus the format of a character string is:




The address of a string specifies the first character of a string. Thus "XYZ" is represented:




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.

### 4.9 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.

All bytes of a trailing numeric string, except the least significant digit byte, must contain an ASCII decimal digit character (0-9). The representation for the high order 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 |
| 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 significant 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 compilers of many 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 valid representations of the digit and sign in each of the later two formats is shown in Table 4-1.

**Table 4-1**
**Representation of Least Significant Digit and Sign**

| | Zoned Numeric Format | | | Overpunch Format | | | |
|---|---|---|---|---|---|---|---|
| digit | decimal | hex | ASCII char. | decimal | hex | ASCII char. norm | alt. |
| 0 | 48 | 30 | 0 | 123 | 7B | { | [ ? |
| 1 | 49 | 31 | 1 | 65 | 41 | A | a |
| 2 | 50 | 32 | 2 | 66 | 42 | B | b |
| 3 | 51 | 33 | 3 | 67 | 43 | C | c |
| 4 | 52 | 34 | 4 | 68 | 44 | D | d |
| 5 | 53 | 35 | 5 | 69 | 45 | E | e |
| 6 | 54 | 36 | 6 | 70 | 46 | F | f |
| 7 | 55 | 37 | 7 | 71 | 47 | G | g |
| 8 | 56 | 38 | 8 | 72 | 48 | H | h |
| 9 | 57 | 39 | 9 | 73 | 49 | I | i |
| −0 | 112 | 70 | p | 125 | 7D | } | ] ! : |
| −1 | 113 | 71 | g | 74 | 4A | J | j |
| −2 | 114 | 72 | r | 75 | 4B | K | k |
| −3 | 115 | 73 | s | 76 | 4C | L | l |
| −4 | 116 | 74 | t | 77 | 4D | M | m |
| −5 | 117 | 75 | u | 78 | 4E | N | n |
| −6 | 118 | 76 | v | 79 | 4F | O | o |
| −7 | 119 | 77 | w | 80 | 50 | P | p |
| −8 | 120 | 78 | x | 81 | 51 | Q | q |
| −9 | 121 | 79 | y | 82 | 52 | R | 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 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. Thus "123" is represented:

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

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 | |
| 4 | C | : A+2 |

## 4.10 LEADING SEPARATE NUMERIC STRING

A leading separate numeric string is a contiguous sequence of bytes in memory. A leading separate numeric string 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 sign of a separate leading numeric string is stored in a separate byte. Valid sign bytes are:

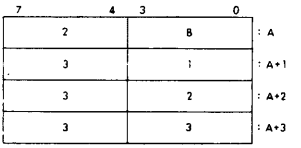
| sign | decimal | hex | ASCII character |
|---|---|---|---|
| + | 43 | 2B | + |
| + | 32 | 20 | <blank> |
| — | 45 | 2D | — |

The preferred representation for "+" is ASCII "+". All subsequent bytes contain an ASCII digit character:

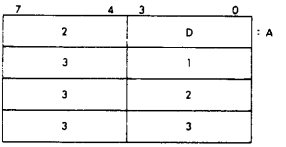
| digit | decimal | hex | ASCII character |
|---|---|---|---|
| 0 | 48 | 30 | 0 |
| 1 | 49 | 31 | 1 |
| 2 | 50 | 32 | 2 |
| 3 | 51 | 33 | 3 |
| 4 | 52 | 34 | 4 |
| 5 | 53 | 35 | 5 |
| 6 | 54 | 36 | 6 |
| 7 | 55 | 37 | 7 |
| 8 | 56 | 38 | 8 |
| 9 | 57 | 39 | 9 |

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 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. Thus "+123" is:




and "—123" is:




## 4.11 PACKED DECIMAL STRING

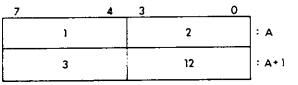
A packed decimal string is a contiguous sequence of bytes in memory. A packed decimal string is specified by two attributes: the address A of the first byte of the string and a length L that is the number of digits in the string and NOT the length of the string in bytes. The bytes of a packed decimal string are divided into two 4-bit fields (nibbles) that must contain decimal digits except the low nibble (bits 3:0) of the last (highest addressed) 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 |

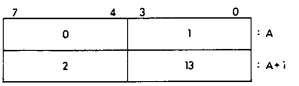
The preferred sign representation is 12 for "+" and 13 for "—". The length L is the number of digits in the packed decimal string (not counting the sign) and must be in the range 0 through 31. When the number of digits is odd, the digits and the sign fit in L/2 (integer part only) + 1 bytes. When the number of digits is even, it is required that

an extra "0" digit appear in the high nibble (bits 7:4) of the first byte of the string. 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. Thus "+123" has length 3 and is represented:




and "—12" has length 2 and is represented:



# CHAPTER 5

# INSTRUCTION FORMATS and ADDRESSING MODES

## 5.1 INTRODUCTION

This chapter describes the addressing modes used in programming the VAX-11 computer. The addressing modes, together with a set of 16 general-purpose registers, provide a convenient method of accessing and manipulating data stored in memory. The addressing modes specify how the selected registers are used to access, manipulate, and store data and instructions.

## 5.2 GENERAL REGISTERS

The VAX-11 general-purpose registers can be used with an instruction in any of the following ways:

- As accumulators. The data to be processed is contained in the register.
- As pointers. The contents of the register are the address of the operand, rather than the operand itself. 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. Automatically stepping forward 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 and are described in subsequent paragraphs in this chapter.
- As index registers. When used as an index register, an offset is generated and is added to the base operand address to yield the indexed location. This is described under Index Mode addressing in this chapter.

One of the general-purpose registers is designated a stack pointer and provides temporary storage for data which is frequently accessed. In the VAX-11 any register can be used as a stack pointer under program control; however, certain instructions associated with subroutine linkage and interrupt service (both of which require storage of linkage information) automatically use register R14 as a "hardware stack pointer." For this reason, R14 is frequently referred to as the "SP". The stack pointer addresses decrease as items are added to the stack. This is conveniently done by decrementing the address and "pushes" data on the stack. This is referred to as autodecrement addressing. The stack pointer addresses increase as items are removed from the stack. This is conveniently done by incrementing the address and "pops" data from the stack. This is referred to as autoincrement addressing. 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 the 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.

## 5.3 INSTRUCTION FORMAT

The VAX-11 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. The general instruction format is shown in Figure 5-1. Each instruction consists of an opcode followed by 0 to 6 operand specifiers whose number and type depend on the opcode. Every operand specifier is 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. The operand type includes both the access type and the data type. Figure 5-2 shows several examples of VAX-11 instruction formats.

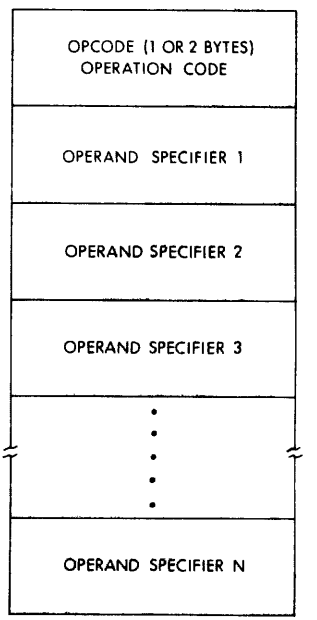



Figure 5-1 General VAX-11 Instruction Format

### 5.3.1 Assembler Notation

The radix of the assembler is in decimal notation. 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 expressed as a hexadecimal number, it would be

MOVW #^X 3456, —(SP).

Examples of hexadecimal numbers and conversion between hex and decimal are provided in Appendix A.

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 | ADDL3 | OPCODE |
| 2 | (SP)+ | OPERAND SPECIFIER 1 |
| 3 | R4 | OPERAND SPECIFIER 2 |
| 4 | (R5) | OPERAND SPECIFIER 3 |

Figure 5-2 Examples of Instruction Format

### 5.3.2 Operation Code (OPCODE)

Each VAX-11 instruction contains an opcode which specifies the desired operation to be performed. The opcode may be one or two bytes long, depending on the instruction. The presently available instruction set only uses a one-byte opcode. Figure 5-3 shows the opcode format.

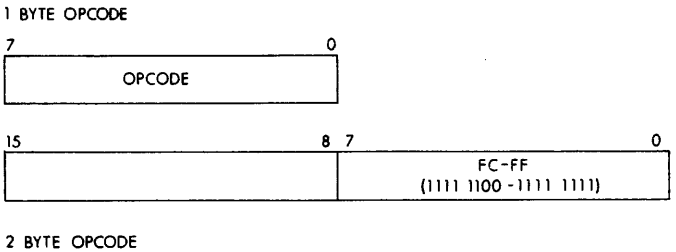



Figure 5-3 Opcode Format

### 5.3.3 Operand Types

The operand types in an instruction specify how the operand associated with an instruction is used. An instruction may have no operands, a single operand or multiple operands. The 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—32-bits
- Floating—32-bit single-precision floating point (same as longword for addressing mode considerations).
- Quad word—64-bit
- Double—64-bit double-precision floating point (same as quad word for addressing mode considerations).

An operand may be accessed in one of the following ways:

- Read—The specified operand is read only.
- Write—The specified operand is written only.
- Modify—The specified operand is read, may or may not be modified and is written.
- 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.
- Variable field—If just Rn is specified, the field is in the general register R[n] or in registers R[n+1] ' R[n] (i.e., registers R[n+1] concatenated with R[n]). Otherwise, address calculation occurs until the actual address of the operand is obtained. This address specified the base to which the field position (offset) is applied.
- 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.

### 5.3.4 Operand Specifier

An operand specifier gives the information needed to locate the operand. For the literal modes, the operand specifier actually includes the value of the operand. Every operand specifier (except branch operands) has the same format and interpretation. The format includes a field that is the address mode. Depending on the mode, this field is 2, 4, or 8 bits. Most address modes include additional information. Depending on the mode up to two register designators are included.

The specifier can also include a displacement address to some location other than the base-register memory location; or the specifier extension can contain immediate data or an absolute address.

## 5.4 ADDRESSING MODES

VAX-11 addressing can be broadly divided into general mode addressing and branch addressing. The two types of branch addressing are designated byte displacement and word displacement. Section 5.5 describes the general mode addressing and Section 5.8 describes branch mode addressing.

Table 5-1 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 reserved addressing mode 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 description of each address mode is an example of how the mode is implemented. The examples show the opcode and operand type notation (opcode src.rx, for example). The src designates source. The r designates that only a read to the source can occur and the x indicates any one of the available data types according to the instruction opcode.

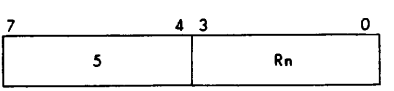
## 5.5 GENERAL MODE ADDRESSING

### 5.5.1 Register Mode

**Assembler Syntax:** Rn

**Mode Specifier:** 5

**Operand Specifier Format:**



R[n+1] ' R[n]. The operand is the contents of Rn for quad, double floating and certain field operands used in the variable bit length field instructions.

Operand = Rn if one register, or
R[n+1] ' R[n] if two 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. If PC is used in a write operand that takes two registers, the contents of R0 is also 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

| R0 | | | | | | | |
|---|---|---|---|---|---|---|---|
| C | 0 | A | 0 | 3 | 4 | 1 | 2 |

| R1 | | | | | | | |
|---|---|---|---|---|---|---|---|
| 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 contents 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 register R2 is unaffected.

**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 |

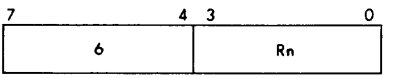
D — displacement
i — any indexable addressing mode
— — logically impossible
f — reserved addressing mode fault
p — Program Counter addressing
u — Unpredictable
uq — Unpredictable for quad and double (and field if position + size greater than 32)
ux — Unpredictable for index register same as base register
y — yes, always valid addressing mode
r — read access
m — modify access
w — write access
a — address access
v — field access

### 5.5.2 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

| | ADDRESS SPACE |
|---|---|
| 00001010 | AB |
| 00001011 | CD |
| 00001012 | EF |
| 00001013 | 12 |
| 00001014 | 34 |
| 00001015 | 56 |
| 00001016 | 76 |
| 00001017 | 65 |

R4

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 1 | 0 | 1 | 0 |

AFTER INSTRUCTION EXECUTION

| | ADDRESS SPACE |
|---|---|
| 00001010 | 00 |
| 00001011 | 00 |
| 00001012 | 00 |
| 00001013 | 00 |
| 00001014 | 00 |
| 00001015 | 00 |
| 00001016 | 00 |
| 00001017 | 00 |

R4

| 0 | 0 | 0 | 0 | 1 | 0 | 1 | 0 |
|---|---|---|---|---|---|---|---|

MACHINE CODE: ASSUME STARTING LOCATION 00003000

| | | |
|---|---|---|
| 00003000 | 7C | OPCODE FOR CLEAR QUAD INSTRUCTION |
| 00003001 | 64 | OPERAND SPECIFIER FOR REGISTER DEFERRED MODE, R4 |
| 00003002 | | |

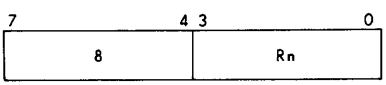
This example shows a Clear Quad instruction using Register Deferred Mode. Register R4 contains the address of the operand and the instruction specifies that this address plus the following seven byte addresses are to be cleared.

### 5.5.3 Autoincrement Mode

**Assembler Syntax:** (Rn)+

**Mode Specifier:** 8

**Operand Specifier Format:**




**Description:** In autoincrement mode addressing, the contents of 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 floating and 8 for quad word or double floating) is added to the contents of register Rn and the contents of Rn is replaced by the result. This mode provides for automatic stepping of a pointer through sequential elements of a table of operands. It assumes the contents of the selected general register to be the address of the operand. 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 completely general and may be used for a variety of purposes.

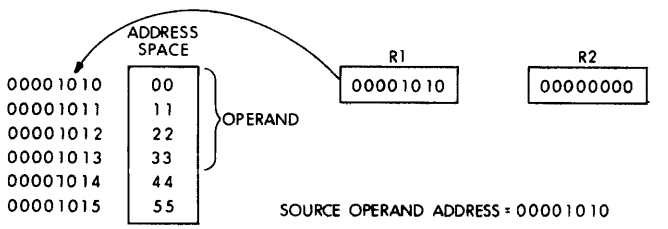
**Special Comments:** If the PC is used as the general register, this addressing mode is designated immediate mode and has special syntax which is described in paragraph 5.7.1.

**EXAMPLE:** AUTOINCREMENT MODE, MOVE LONG INSTRUCTION

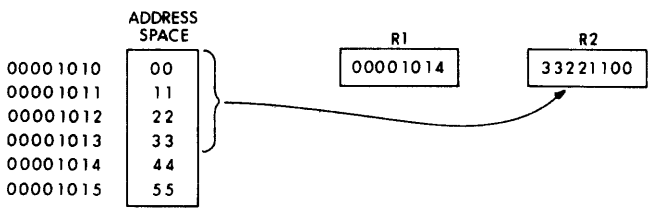
**Instruction Format:** MOVL (R1)+, R2

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

BEFORE INSTRUCTION EXECUTION




AFTER INSTRUCTION EXECUTION




MACHINE CODE: ASSUME STARTING LOCATION 3000

| | | |
|---|---|---|
| 00003000 | D0 | OPCODE FOR MOVE LONG WORD INSTRUCTION |
| 00003001 | 81 | AUTOINCREMENT MODE, REGISTER R1 |
| 00003002 | 52 | REGISTER MODE, REGISTER R2 |
| 00003003 | | |

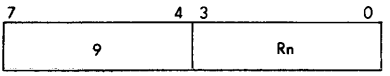
This example shows a Move Long instruction using autoincrement mode. The contents of register R1 is the effective address of the source operand. The operand is a 32-bit longword and, therefore, four bytes are transferred to register R2. R1 is then incremented by 4 since the instruction specifies a longword data type.

### 5.5.4 Autoincrement Deferred Mode

**Assembler Syntax:** @ (Rn)+

**Mode Specifier:** 9

**Operand Specifier Format:**



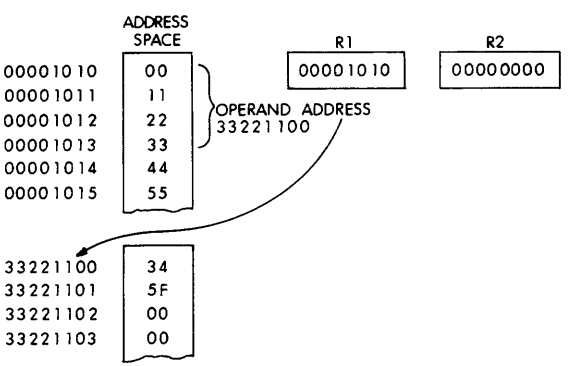

**Description:** In autoincrement deferred addressing, register Rn contains a longword address which is a pointer to the operand address. After the operand address has been determined, 4 is added to the contents of register Rn and the contents of register Rn is replaced with the result. The quantity 4 is used since there are 4 bytes in an address.

**Special Comments:** If the PC is used as the general register, this addressing mode is designated absolute mode and is described in paragraph 5.7.2.

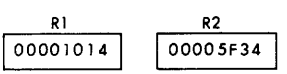
**EXAMPLE:** AUTOINCREMENT DEFERRED MODE, MOVE WORD INSTRUCTION

**Instruction Format:** MOVW @(R1)+, R2

BEFORE INSTRUCTION EXECUTION




AFTER INSTRUCTION EXECUTION



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 |

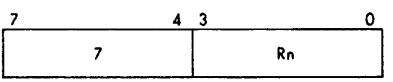
This example shows a Move Word instruction using autoincrement deferred mode. The contents of register R1 is a pointer to the operand address. Since a word length instruction is specified, the byte at the effective address and the byte at the effective address plus 1 are loaded into the low-order half of register R2 with the upper half of R2 unspecified. R1 is then incremented by 4 since it contains a 32-bit address.

### 5.5.5 Autodecrement Mode

**Assembler Syntax:** —(Rn)

**Mode Specifier:** 7

**Operand Specifier Format:**



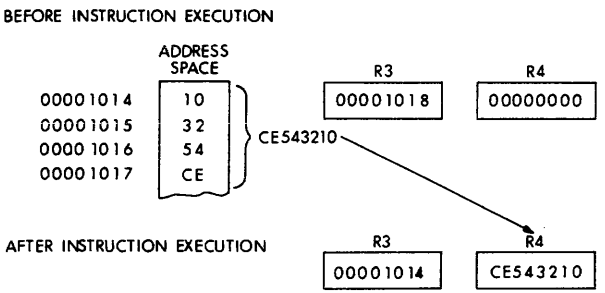

The contents of Rn are decremented and then used as the address of the operand.

**Description:** With autodecrement mode, the size of the operand in bytes (1 for byte, 2 for word, 4 for longword or floating and 8 for quad word or double) is subtracted from the contents of register Rn and the contents of register Rn are replaced by the result. The updated contents of register Rn is the address of the operand. The contents of the selected general register are decremented and then used as 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 unpredictable.

**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 |

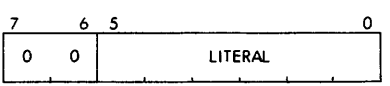
This example shows a Move Long instruction using autodecrement mode. The contents of register R3 is decremented according to the data type specified in the opcode (4 in this example because a longword is used). The updated contents of register R3 is then used as the address of the operand. The instruction causes the operand to be fetched and loaded into register R4.

### 5.5.6 Literal Mode

**Assembler Syntax:** S^# literal

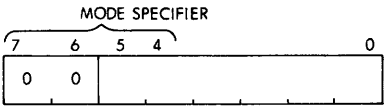
**Mode Specifier:** 0, 1, 2 or 3
(depending on literal value specified)

**Operand Specifier Format:**

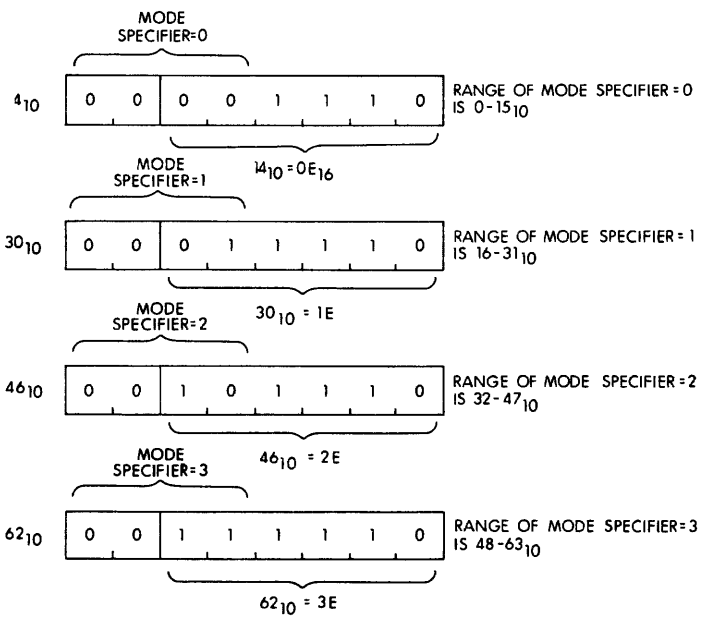



The S^ syntax can be used to force literal mode; otherwise, the assembler will force literal or immediate mode, whichever is more appropriate.

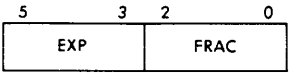
**Description:** Literal mode addressing provides an efficient means of specifying integer constants in the range from 0 to 63 (decimal). This is called short literal. Literal values above 63 can be obtained by immediate mode (autoincrement mode using the PC). For short literal operands, the format is:



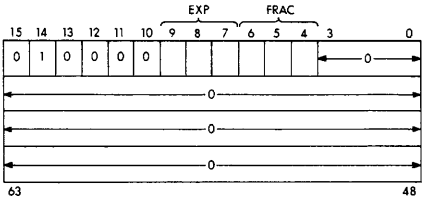

Bits 7 and 6, however, are always set to zero. The following examples show some short literals; the literals are 14, 30, 46, and 62.




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 fraction.




The 3-bit EXP field and 3-bit FRAC field are used to form a floating or double-floating operand as follows:




**NOTE**

Bits 32-63 are not present in single-precision floating point operands.

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 it is desired to express the floating point literal of 12. Table 5-2 shows this decimal literal of 12 to be represented by a fraction of 4 and an exponent of 4.

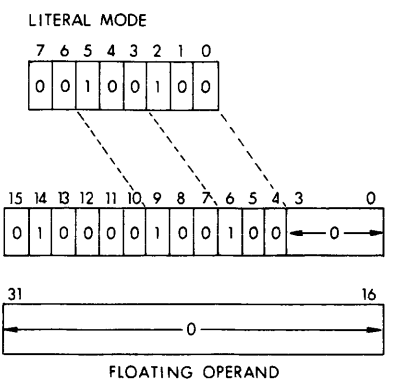
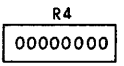


**Table 5-2 Floating Literals**

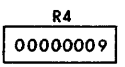
| Exponent | FRACTION | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| 0 | 1/2 | 9/16 | 5/8 | 11/16 | 3/4 | 13/16 | 7/8 | 15/16 |
| 1 | 1 | 1 1/8 | 1 1/4 | 1 3/8 | 1 1/2 | 1 5/8 | 1 3/4 | 1 7/8 |
| 2 | 2 | 2 1/4 | 2 1/2 | 2 3/4 | 3 | 3 1/4 | 3 1/2 | 3 3/4 |
| 3 | 4 | 4 1/2 | 5 | 5 1/2 | 6 | 6 1/2 | 7 | 7 1/2 |
| 4 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
| 5 | 16 | 18 | 20 | 22 | 24 | 26 | 28 | 30 |
| 6 | 32 | 36 | 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

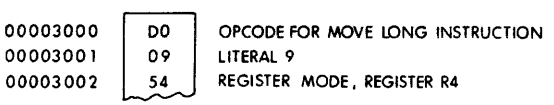
BEFORE INSTRUCTION EXECUTION




AFTER INSTRUCTION EXECUTION




MACHINE CODE: ASSUME STARTING LOCATION 00003000



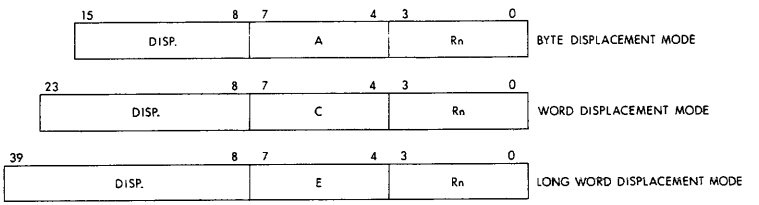

This example shows a Move Long instruction using literal mode. The literal 9 is transferred to register R4 as a result of the instruction.

### 5.5.7 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:**

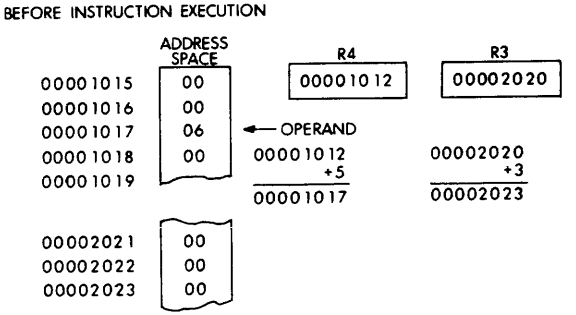



**Description:** In displacement mode addressing, the displacement (after being sign extended to 32 bits if it is a byte or word) is added to the contents of register Rn and the result is the operand address. This mode is the equivalent of index mode in the PDP-11 series.

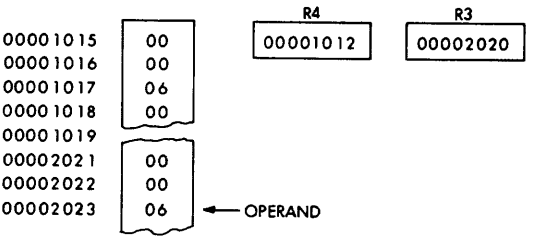
The VAX-11 architecture provides for an 8-bit, 16-bit or 32-bit offset. Since most program references occur within small discrete portions of the address space, a 32-bit offset is not always necessary and the 8- and 16-bit offsets will result in substantial economies of space (that is, fewer bits are required).

If the PC is used as the general register, this mode is called relative mode and is described in paragraph 5.7.3.

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



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 contents of Register 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 Register R3.

### 5.5.8 Displacement Deferred Mode

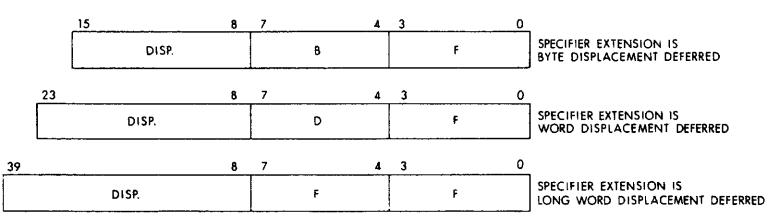
**Assembler Syntax:**

@ R(Rn)
@ B^D(Rn) byte displacement deferred
@ W^D(Rn) word displacement deferred
@ L^D(Rn) long word displacement deferred

**Mode Specifier:**

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

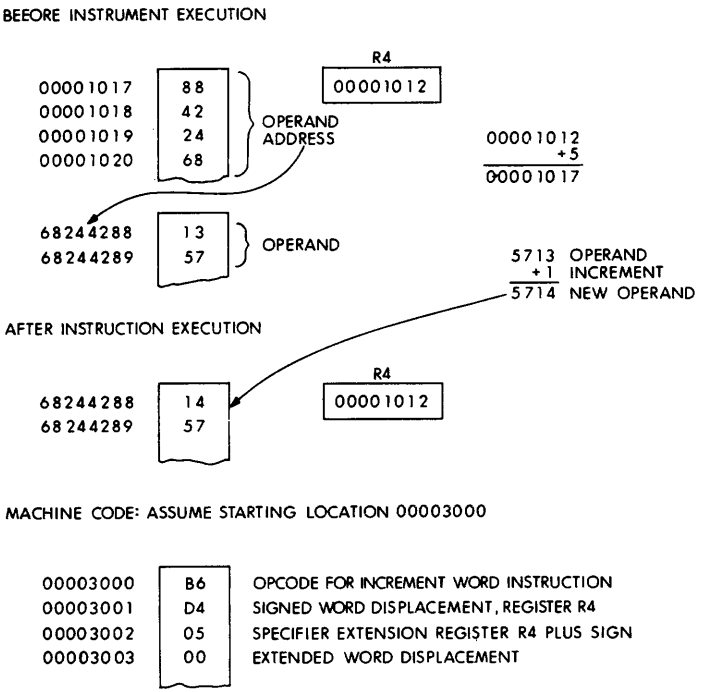
**Operand Specifier Format:**



**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 contents of the selected general register 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 and is described in paragraph 5.7.4.

**EXAMPLE:** DISPLACEMENT DEFERRED MODE, INCREMENT WORD INSTRUCTION
INCW @W^5(R4)




MACHINE CODE: ASSUME STARTING LOCATION 00003000

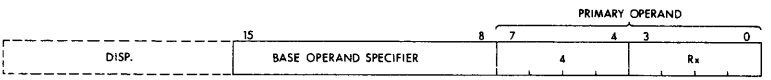
| | | |
|---|---|---|
| 00003000 | B6 | OPCODE FOR INCREMENT WORD INSTRUCTION |
| 00003001 | D4 | SIGNED WORD DISPLACEMENT, REGISTER R4 |
| 00003002 | 05 | SPECIFIER EXTENSION REGISTER R4 PLUS SIGN |
| 00003003 | 00 | EXTENDED WORD DISPLACEMENT |

## 5.6 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 floating, and 8 for quad word or double). 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. The VAX-11 architecture provides for context indexing whereby the number in the index register is shifted left by the context of the data type specified (once for byte, twice for word, three times for longword, four times for quadword). 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:

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

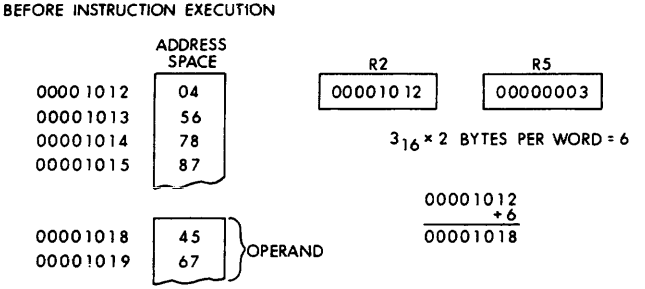
ing "indexed" to the addressing mode of the base operand specifier. The general register is designated Rn and the indexed register is Rx.

**Table 5-3**
**Index Mode Addressing**

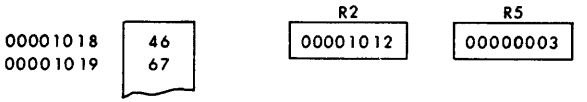
| MODE | ASSEMBLER NOTATION |
|---|---|
| Register deferred index | (Rn) [Rx] |
| Autoincrement indexed | (Rn) + [Rx] |
| Immediate indexed | I# constant [Rx] which is recognized by assembler but is not generally useful. Operand address is independent of value of constant. |
| Autoincrement deferred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



AFTER INSTRUCTION EXECUTION



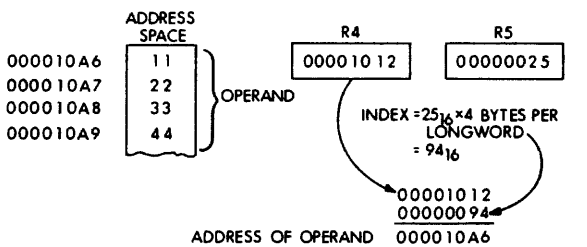

ASSEMBLY CODE: ASSUME STARTING LOCATION 00003000

| | | |
|---|---|---|
| 00003000 | B6 | OPCODE FOR INCREMENT WORD INSTRUCTION |
| 00003001 | 45 | INDEX MODE, REGISTER R5 |
| 00003002 | 62 | REGISTER DEFERRED MODE, REGISTER R2 |

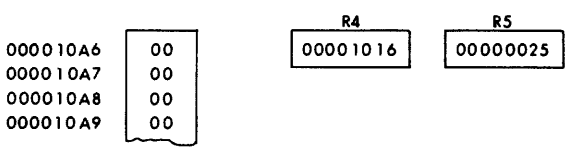
This example shows an Increment Word instruction using register deferred index addressing. The base operand address is evaluated. This location is indexed by 6 since the value (3) in the index register is multiplied by the word data size of 2.

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

BEFORE INSTRUCTION EXECUTION




AFTER INSTRUCTION EXECUTION

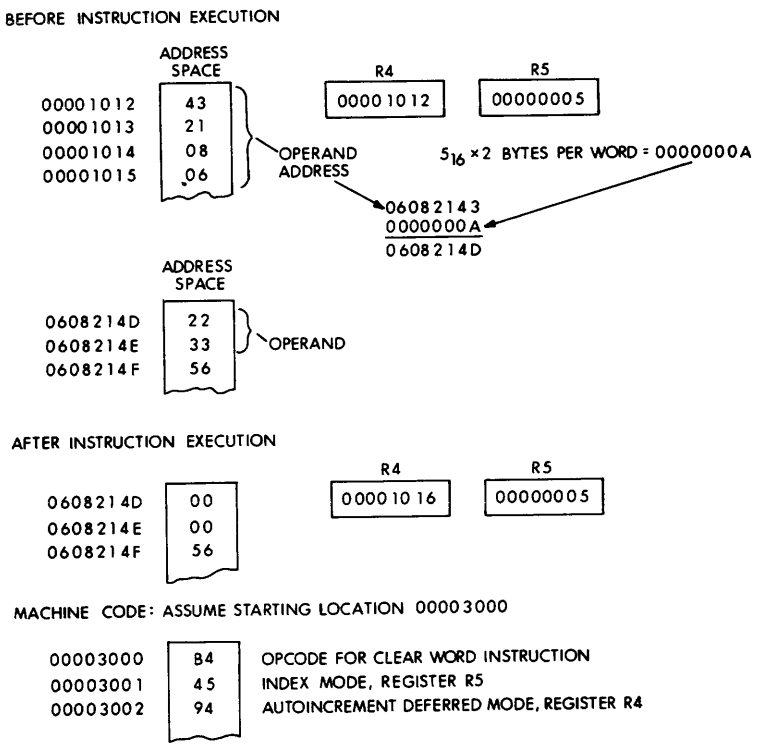



MACHINE CODE: ASSUME STARTING LOCATION 00003000

| | | |
|---|---|---|
| 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 register R4. This value is indexed by the quantity in register R5 multiplied by the data size. This location, plus the next three, are cleared since a clear longword instruction is specified.

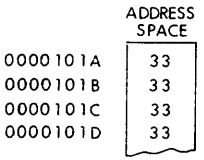
**EXAMPLE:** AUTOINCREMENT DEFERRED INDEX MODE, CLEAR WORD INSTRUCTION
CLRW @(R4) + [R5]

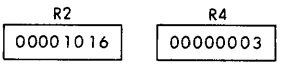



This example shows a Clear Word instruction using the autoincrement deferred indexing mode. Register R4 contains the address of the operand address. The index value of A is obtained by multiplying the contents (5) of the index register by the context of the data type which is 2. The calculated word address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

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

BEFORE INSTRUCTION EXECUTION

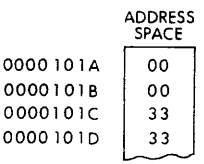


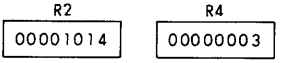




$3_{16}$ × 2 BYTES PER WORD = 6 (INDEX)

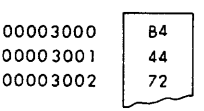
```
00001016
00000002  DECREMENT BY 2
00001014  OPERAND ADDRESS
00000006  INDEX VALUE
0000101A  INDEXED OPERAND ADDRESS
```

AFTER INSTRUCTION EXECUTION



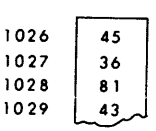

MACHINE CODE: ASSUME STARTING LOCATION 00003000

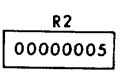



This example shows a Clear Word instruction using autodecrement indexed mode. The contents of register R2 are predecremented and the indexed value is calculated as 6. Since a clear word instruction is specified, two bytes are cleared.

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

BEFORE INSTRUCTION EXECUTION

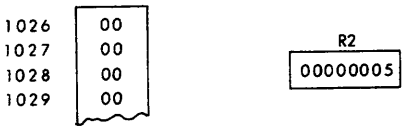



$5_{16} \times 4 = 14_{16}$

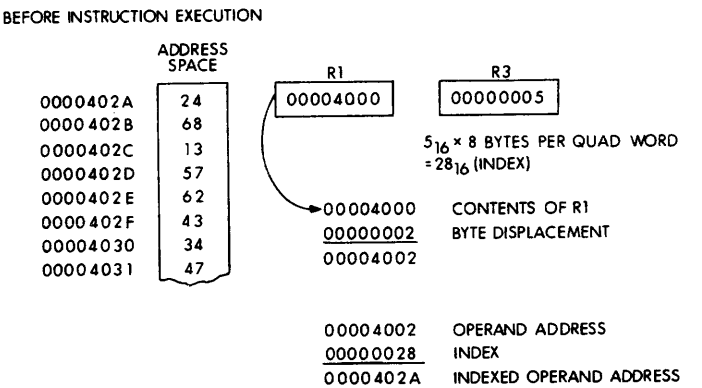
```
00001012
00000014
0000 1026
```

AFTER INSTRUCTION EXECUTION



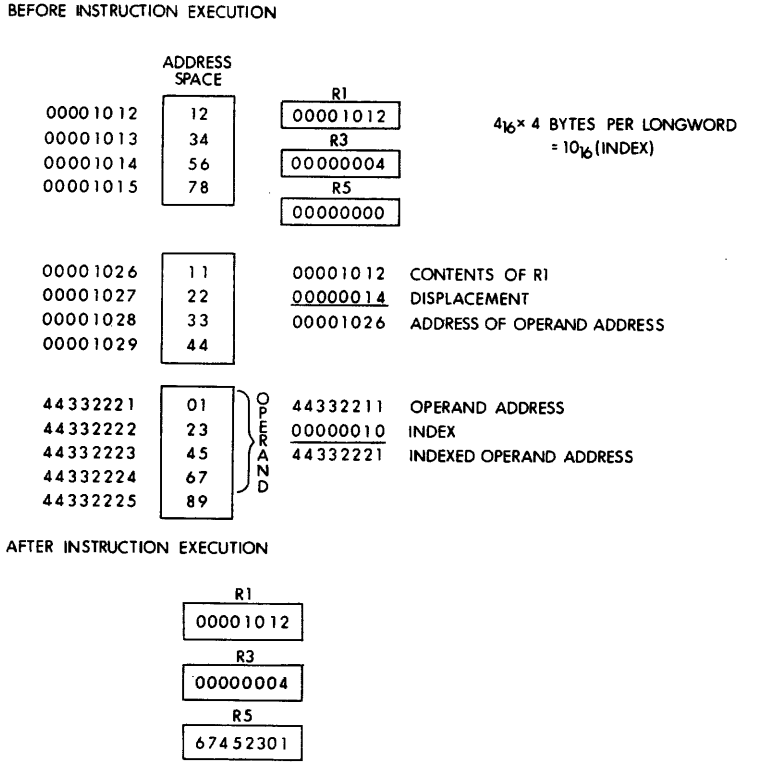

This example shows a Clear Longword instruction using absolute indexed mode. The base of 00001012 is indexed by R2 which contains 5. Since a longword data type is specified, 5 x 4 = $14_{16}$ which becomes the index value. This value is added to 00001012 yielding 0000-1026. This is the operand address 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 2 is added to the contents of register 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 register 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 register R5 as shown.

## 5.7 PROGRAM COUNTER ADDRESSING

Register R15 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, etc.

The PC can be used with all of the VAX-11 addressing modes except register or index mode since the results will be unpredictable. The following four 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, except PC is used as general register.

**Absolute mode**—same as autoincrement deferred mode, except PC is used as general register.

**Relative mode**—same as displacement mode, except PC is used as general register.

**Relative deferred mode**—same as displacement deferred mode except PC is used as general register.

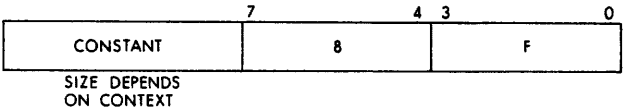
When a standard program is available for different users, it is often helpful to be able to load it into different areas of memory and run it there. The VAX-11/780 can accomplish the relocation of a program very efficiently through the use of position independent code (PIC). If an instruction and its 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.

### 5.7.1 Immediate Mode

**Assembler Syntax:** I^# operand

**Mode Specifier:** 8

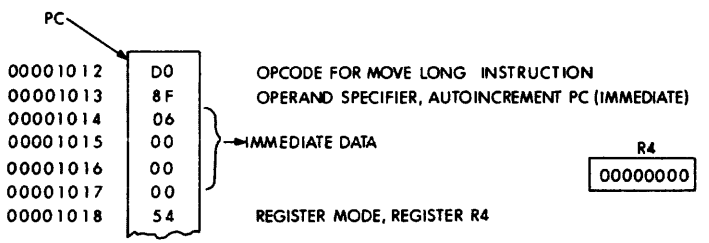
**Operand Specifier Format:**



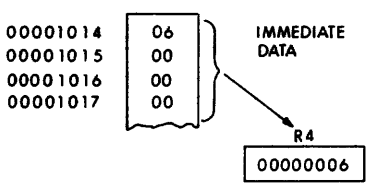

**Description:** The immediate addressing mode is autoincrement mode when the PC is used as the general register. The contents of the location following the addressing mode is immediate data.

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

BEFORE INSTRUCTION EXECUTION




AFTER INSTRUCTION EXECUTION



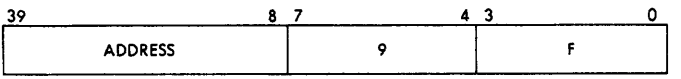

This example shows a Move Long instruction using immediate mode. The immediate data (00000006) following the opcode and operand specifier is moved to the contents of R4.

### 5.7.2 Absolute Mode

**Assembler Syntax:** @#location

**Mode Specifier:** 9

**Operand Specifier Format:**

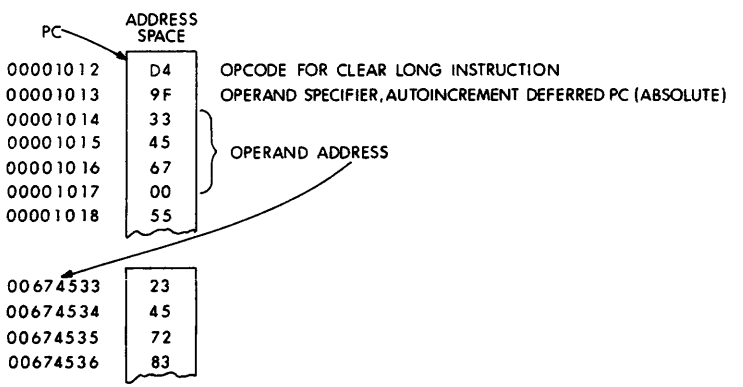



**Description:** This mode is autoincrement deferred using the PC as the general register. The contents of the location following the addressing mode are taken as the operand address. This is interpreted as an absolute address (an address that remains constant no matter where in memory the assembled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

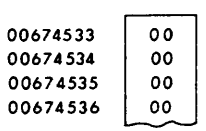
**EXAMPLE:** ABSOLUTE MODE, CLEAR LONG INSTRUCTION

```
CLRL @#^674533
```

BEFORE INSTRUCTION EXECUTION




AFTER INSTRUCTION EXECUTION



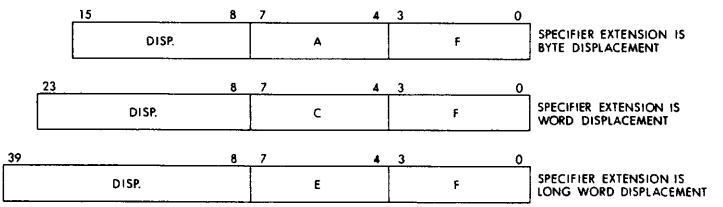

This example shows a Clear Longword instruction using the absolute addressing mode. This instruction causes the location(s) following the addressing mode to be taken as the address of the operand, and is 00674533, in this case. The longword operand associated with this address is cleared.

### 5.7.3 Relative Mode

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

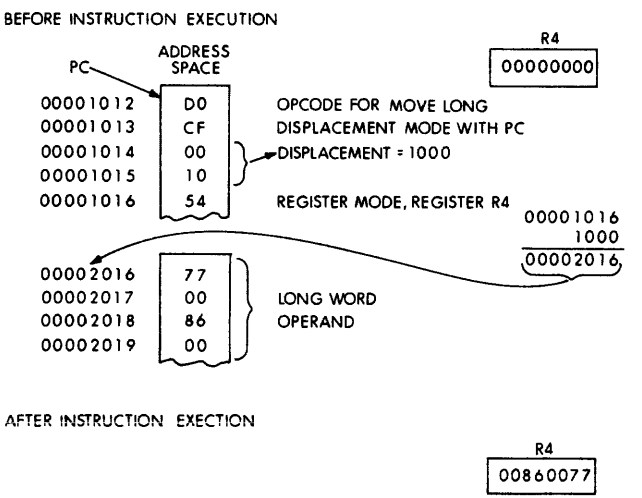
**Mode Specifier:** A (Byte), C (Word), E (Longword)

**Operand Specifier Format:**




**Description:** This mode is the displacement mode 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 0000-1000, 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.

### 5.7.4 Relative Deferred Mode

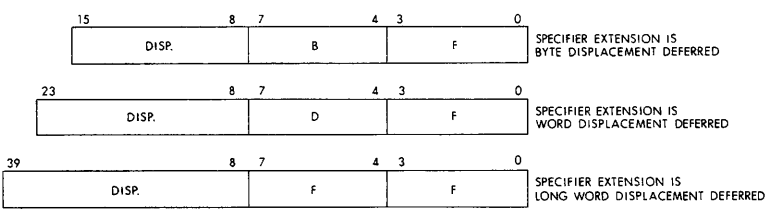
**Assembler Syntax:**

@ B^D—Byte displacement deferred
@ W^D—Word displacement deferred
@ L^D—Longword displacement deferred

**Mode Specifier:**

A (byte deferred), C (word deferred), E (longword deferred)

**Operand Specifier Format:**

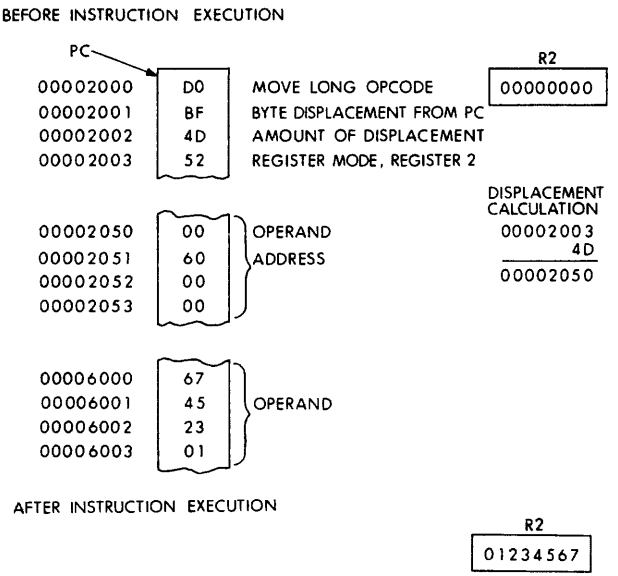



**Description:**

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



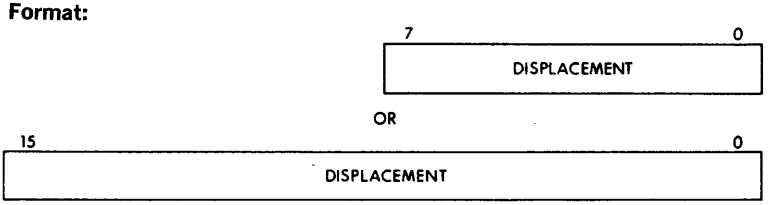

This example shows a Move Long instruction where 00002050 represents the address of the operand. A byte displacement would be selected by the assembler since the displacement is within 128 (decimal) addressable bytes. When the displacement is evaluated, the program counter is pointing to 00002003. The 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.

## 5.8 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 contents of the PC. The updated contents 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, #'0  ;Compare C and ASCII representation of digit 0
BLSSU NOT    ;Branch to location NOT if less
              than an unsigned 0.
CMPB C, #'9  ;Compare C and ASCII representation of digit 9.
BGTRU NOT    ;Branch to location NOT if greater
              than an unsigned 9.
```

**EXAMPLE:** BRANCH ON BIT

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

# CHAPTER 6

# INTEGER AND FLOATING POINT INSTRUCTIONS

## 6.1. INSTRUCTION SET OVERVIEW

A major goal of the VAX-11 architecture is to provide an instruction set that is symmetrical with respect to data types. For example, the ADD instruction occurs for each of the five integer and floating point data types. These instructions are available symmetrically for each of the three integer data types (byte, word, and longword) and the two floating point data types (floating and double). The symmetric operations include data movement, data conversion, data testing and computation. Thus both assembly language programmers and compilers can choose the instruction to use independently of the data type.

To simplify the understanding of the instruction set, the instruction mnemonics are formed by combining an operation prefix with a data type suffix. The convert instructions are formed by adding suffixes for both the source and destination data types. The computation instructions include a further suffix to indicate the choice between two-operand and three-operand instructions. The special instruction mnemonics have been chosen for similarity. Figures 6-1 and 6-2 show the instruction mnemonics. For example, a move word instruction has the mnemonic MOVW while a move floating instruction has the mnemonic MOVF.

| INSTRUCTION | DATA TYPE | NO. OF OPERANDS |
|---|---|---|
| MOVe<br>CLearR<br>Move NEGative<br>CoMPare<br>TeST | Byte<br>Word<br>Longword<br>Floating<br>Double | 2 operand |
| Move Complement<br>BIt Test | Byte<br>Word<br>Longword | |
| ConVerT | Byte<br>Word<br>Longword<br>Floating<br>Double<br>to<br>Byte<br>Word<br>Longword<br>Floating<br>Double | 2 operand |
| | except BB, WW, LL, FF, DD | |
| ADD<br>SUBtract<br>MULtiply<br>DIVide | Byte<br>Word<br>Longword<br>Floating<br>Double | 2 operand<br>3 operand |

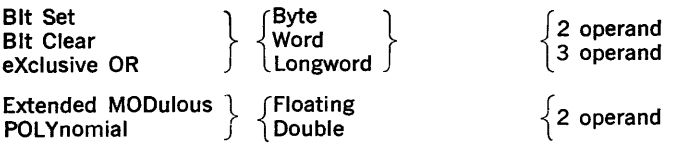



Figure 6-1 Integer & Floating Instructions

| INSTRUCTION | DATA TYPE |
|---|---|
| PUSH Longword | |
| INCrement, DECrement | Byte, Word, Longword |
| MOVe, CLeaR | Quadword |
| MOVe Zero-extend | Byte to Word, Byte to Longword, Word to Longword |
| ConVerT Round | Floating, Double to longword |
| ADd Aligned Word under memory Interlock<br>ADd With Carry<br>SuBtract With Carry<br>Extended MULtiply<br>Extended DIVide | |
| Arithmetic SHift | Longword, Quadword |
| ROTate Longword | |

Figure 6-2 Optimizations and Special Operations

The move operations are simple move, clear (move zero), arithmetic negate, and logical complement. Move and clear are also available for the quadword data type. 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-11 includes a complete set of converts from each of the five data types to each of the other types. 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.

VAX-11 computation instructions for all five data types 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. The arithmetic and logical computation instructions are available in both two and three operand forms for each applicable data type. The two operand form takes as input the value of each operand and stores the result as a modification to the second operand. 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 by one. VAX-11 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-11 includes special floating point instructions for modules (range reduction) and polynomial calculation to aid in the implementation of mathematical functions. VAX-11 also includes shift and rotate instructions.

## 6.2 FLOATING POINT INSTRUCTIONS

In order to be consistent with the floating point instruction set which faults on reserved operands (see Chapter 4), software implemented floating point functions (e.g., the absolute function) should verify that the input operand(s) is (are) not reserved. An easy way to do this is a floating or double floating move or test of the input operand(s).

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 register Rn is used as both a floating point operand or either part of a double floating input operand (i.e., a .rf, .rd, .mf, or .md 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.

### 6.2.1 Introduction

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

$$(+ \text{ or } -)\ (2^{**}K)^{*}f,$$

where K is an integer and f is a non-negative fraction. For a non-vanishing number, K and f are uniquely determined by imposing the condition

$$\tfrac{1}{2} \text{ LEQ } f \text{ LSS } 1.$$

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

The VAX-11 floating point data formats are derived from this mathematical representation for floating point numbers. Two types of floating point data are provided. Single precision, or floating, data is 32 bits long. Double precision, or double, data is 64 bits long. Sign magnitude notation is used, as follows:

1. Non-zero floating point numbers:

   The most significant bit of the floating point data is the sign bit: 0 for positive, and 1 for negative.

   The fractional factor f is assumed normalized, so that its most significant bit must be 1. This 1 is the "hidden" bit: it is not stored in the data word, but the hardware restores it before carrying out arithmetic operations. The floating and double data types use 23 and 55 bits, respectively, for f, which with the hidden bit, imply effective significance of 24 bits and 56 bits for arithemtic operations.

   Eight bits are reserved for the storage of the exponent K in excess 128 notation. Thus exponents from $-128$ to $+127$ could be represented, in biased form, by 0 to 255. For reasons given below, a biased EXP of 0 (true exponent of $-128$), is reserved for floating point zero. Thus VAX-11 exponents are restricted to the range $-127$ to $+127$ inclusive, or in excess 128 notation, 1 to 255.

2. Floating point zero:

   Because of the hidden bit, the fractional factor is not available to distinguish between zero and non-zero numbers whose fractional factor is exactly $\frac{1}{2}$. Therefore VAX-11 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 0's, is treated as zero, and this is the format generated by all floating point instructions for which the result is zero.

3. The reserved operands:

   A reserved operand is defined to be any bit pattern with a sign bit of one and a biased exponent of zero. On VAX-11, all floating point instructions generate a fault if a reserved operand is encountered. Since a reserved operand has a biased exponent of zero, it can be (internally) generated only if either overflow or underflow occurs.

### 6.2.2 Accuracy

General comments on the accuracy of the VAX-11 floating point intruction set are presented here. The descriptions of some individual instructions include additional details on the accuracy at which they operate.

An instruction is defined to be exact if its result, extended on the right by an infinite sequence of zeros, is identical to that of an infinite precision calculation involving the same operands. The a priori accuracy of the operands is thus ignored. For all arithmetic operations, except DIV, a zero operand implies that the instruction is exact. The same state-

ment holds for DIV if the zero operand is the dividend. But if it is the divisor, division is undefined and the instruction traps.

For non-zero floating point operands, the fractional factor is binary normalized with 24 or 56 bits for single or double precision, respectively. We show below that for 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 $\frac{1}{2}$ LSB (least significant bit).

Note that an arithmetic result is exact if no non-zero bits are lost in chopping the infinite precision result to the data length to be stored. Chopping is defined to mean that the 24 or 56 high order bits of the normalized fractional factor of a result are stored; the rest of the bits are discarded. 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 one, the rounded result is the chopped result incremented by an LSB (least significant bit).
2. If the rounding bit is zero, 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. The following statements 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, it's magnitude
   - is always less than that of the true result for chopping.
   - is always less than that of the true result for rounding if the rounding bit is zero.
   - is greater than that of the true result for rounding if the rounding bit is one.

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 or 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 ($\frac{1}{2}$) 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 alignment 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 non-zero 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 non-zero 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 is 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, which might make good use of high speed multiplication logic, 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-11 observes all constraints and generates floating point results with an error bound of (½) LSB for all floating point instructions except EMOD and POLY (see EMOD and POLY descriptions.)

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

## 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 Long |
| 7D | MOVQ | Move Quad |
| 50 | MOVF | Move Floating |
| 70 | MOVD | Move Double |

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

**Notes:** On a floating reserved operand fault, the destination operand is unaffected and the condition codes are unpredictable.

## PUSH LONG

**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 Long

**Description:** The long word source operand is pushed on the stack.

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

## 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 Long |
| 7C | CLRQ | Clear Quad |
| D4 | CLRF | Clear Floating |
| 7C | CLRD | Clear Double |

**Description:** The destination operand is replaced by 0.

**Notes:** CLRx dst is equivalent to MOVx^0, dst, but is shorter.

## MOVE NEGATED

**Purpose:** move the arithmetic negation of a scalar quantity

**Operation:** dst ← —src;

**Format:** opcode src.rx, dst.wx

**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 Long |
| 52 | MNEGF | Move Negated Floating |
| 72 | MNEGD | Move Negated Double |

**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.

## MOVE COMPLEMENTED

**Purpose:** move 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 Long |

**Description:** The destination operand is replaced by the ones complement of the source operand.

## 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};
C ← 0

**Exceptions:**
Integer overflow
Floating overflow
Reserved operand

**Opcodes:**

| | | |
|---|---|---|
| 99 | CVTBW | Convert Byte to Word |
| 98 | CVTBL | Convert Byte to Long |
| 33 | CVTWB | Convert Word to Byte |
| 32 | CVTWL | Convert Word to Long |
| F6 | CVTLB | Convert Long to Byte |
| F7 | CVTLW | Convert Long to Word |
| 4C | CVTBF | Convert Byte to Floating |
| 6C | CVTBD | Convert Byte to Double |
| 4D | CVTWF | Convert Word to Floating |
| 6D | CVTWD | Convert Word to Double |
| 4E | CVTLF | Convert Long to Floating |
| 6E | CVTLD | Convert Long to Double |
| 48 | CVTFB | Convert Floating to Byte |
| 68 | CVTDB | Convert Double to Byte |
| 49 | CVTFW | Convert Floating to Word |
| 69 | CVTDW | Convert Double to Word |
| 4A | CVTFL | Convert Floating to Long |
| 4B | CVTRFL | Convert Rounded Floating to Long |
| 6A | CVTDL | Convert Double to Long |
| 6B | CVTRDL | Convert Rounded Double to Long |
| 56 | CVTFD | Convert Floating to Double |
| 76 | CVTDF | Convert Double to 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 | CVTFW | truncated |
| CVTBD | exact | CVTDW | truncated |
| CVTWF | exact | CVTFL | truncated |
| CVTWD | exact | CVTRFL | rounded |
| CVTLF | rounded | CVTDL | truncated |
| CVTLD | exact | CVTRDL | rounded |
| CVTFB | truncated | CVTFD | exact |
| CVTDB | truncated | CVTDF | 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 result.
3. Only CVTDF can result in floating overflow. On floating overflow, the destination operand is replaced by an operand of all 0 bits except for a sign bit of 1 (a reserved operand). N ← 1; Z ← 0; V ← 1; and C ← 0.
4. Only converts with a floating point source operand can result in a reserved operand fault. On a reserved operand fault, the destination operand is unaffected and the condition codes are unpredictable.

**EXAMPLE:** CONVERT FLOATING TO WORD

```
CVTFW work, R0          ;Convert contents of R2
                        ;floating to long, rounding
                        ;store in R3
```

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

**After Instruction Execution:**
Work = 00004410
R0 = 00000090 (hex) (integer 144)

**EXAMPLE:** CONVERT ROUNDED FLOATING TO LONG

```
CVTRFL R2,R3            ;Convert contents of work
                        ;floating to word;
                        ;store in R0
```

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

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

## 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 Long |
| 3C | MOVZWL | Move Zero-Extended Word to Long |

**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 destination 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.

## 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 Long |
| 51 | CMPF | Compare Floating |
| 71 | CMPD | Compare Double |

**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.

## 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 Long |

**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 in incremented. On overflow, the sum operand is replaced by the largest negative integer.
2. INCx sum is equivalent to ADDx #1, sum, but is one byte shorter.

## TEST

**Purpose:** arithmetic compare of a scalar to 0.

**Format:** opcode src.rx

**Operation:** src — 0;

**Condition Codes:**
N ← src LSS 0;
Z ← src 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 Long |
| 53 | TSTF | Test Floating |
| 73 | TSTD | Test Double |

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

**Notes:**

1. TSTx src is equivalent to CMPx src, ^#0, but is shorter.
2. On a floating reserved operand, the condition codes are unpredictable.

## 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 Long 2 Operand |
| C1 | ADDL3 | Add Long 3 Operand |
| 40 | ADDF2 | Add Floating 2 Operand |
| 41 | ADDF3 | Add Floating 3 Operand |
| 60 | ADDD2 | Add Double 2 Operand |
| 61 | ADDD3 | Add Double 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, the sum operand is replaced by 0.
4. On floating overflow, the sum operand is replaced by an operand of all bits 0 except for a sign bit of 1 (a reserved operand). N ← 1; Z ← 0; V ← 1; and C ← 0.

**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)

## 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 overflow);
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 true result.
2. The 2 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.

## ADD ALIGNED WORD INTERLOCKED

**Purpose:** maintain operating system resource usage counts

**Format:** opcode add.rx, sum.mx

**Operation:**
{set interlock};
sum ← sum + add;
{release 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 on other processors in a multiprocessor system. The destination must be aligned on a word boundary. If it is not, a reserved operand fault is taken.

**Notes:** 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.

## SUBTRACT

**Purpose:** perform arithmetic subtraction

**Format:**

```
opcode sub.rx, dif.mx                    2 operand
opcode sub.rx, min.rx, dis.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 Long 2 Operand |
| C3 | SUBL3 | Subtract Long 3 Operand |
| 42 | SUBF2 | Subtract Floating 2 Operand |
| 43 | SUBF3 | Subtract Floating 3 Operand |
| 62 | SUBD2 | Subtract Double 2 Operand |
| 63 | SUBD3 | Subtract Double 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, the difference operand is replaced by 0.
4. On floating overflow, the difference operand is replaced by an operand of all 0 bits except for a sign bit of 1 (a reserved operand). N ← 1; Z ←0; V ← 1; and C ← 0.

**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

## 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 into most significant bit⟩

**Exceptions:** Integer overflow

**Opcodes:**

| | | |
|---|---|---|
| 97 | DECB | Decrement Byte |
| B7 | DECW | Decrement Word |
| D7 | DECL | Decrement Long |

**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 #1, dif, but is one byte shorter.

## 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 2 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.

## 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 Long 2 Operand |
| C5 | MULL3 | Multiply Long 3 Operand |
| 44 | MULF2 | Multiply Floating 2 Operand |
| 45 | MULF3 | Multiply Floating 3 Operand |
| 64 | MULD2 | Multiply Double 2 Operand |
| 65 | MULD3 | Multiply Double 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. On integer overflow, the product operand is replaced by the low order bits of the true result.
2. On a floating reserved operand abort, the product operand is unaffected and the condition codes are unpredictable.
3. On floating underflow, the product operand is replaced by 0.
4. On floating overflow, the product operand is replaced by an operand of all bits 0 except for a sign bit of 1 (a reserved operand). N ← 1; Z ← 0;V ← 1; and C ← 0

**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

```
MULF 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

## EXTENDED MULTIPLY

**Purpose:** perform extended-precision multiplication

**Format:** opcode mulr.rl, muld.rl, add.rl, prod.wg

**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. The product operand is replaced by the final result.

**EXAMPLE:** EXTENDED MULTIPLY

To multiply two quadwords, producing a quadword;

```
    EMUL A, B, C            ;multiply low half
    MULL3 A+4, B, R0        ;high half = A [high] *
    MULL3 A, B+4, R1;       ;B [low]
                            ;+ A [low] * B
                            ;[high]
    ADDL R1, R0             ;(combine)
    TSTL A                  ;if A [low] < 0, need to
    BGEQ 1 0 $              ;compensate for unsigned
    ADDL B, R0              ;bias of 2**32
10$:TSTL B                  ;if B [low] 20, 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]
```

## EXTENDED MULTIPLY AND INTEGERIZE

**Purpose:** perform accurate range reduction of math function arguments

**Format:** opcode mulr.rx, mulrx.rb, muld.rx, int.wl, fract.wx

**Operation:**
int ← integer part of muld * {mulr,mulrx};
frac ← 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 Floating |
| 74 | EMODD | Extended Multiply and Integerize Double |

**Description:** The floating point multiplier extension operand (second operand) is concatenated with the floating point mutiplier (first operand) to gain 8 additional low order fraction bits. The multiplicand operand is multiplied by the extended multiplier operand. After multiplication, the integer portion is extracted and a 32-bit (EMODF) or 64-bit (EMODD) floating point number is formed from the fractional part of the product by truncating extra bits. The multiplication is such that the result is equivalent to the exact product truncated to a fraction field of 32 bits in floating and 64 bits in double. 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 integer and fraction operands are replaced by zero.
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.

## 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 Long 2 Operand |
| C7 | DIVL3 | Divide Long 3 Operand |
| 46 | DIVF2 | Divide Floating 2 Operand |
| 47 | DIVF3 | Divide Floating 3 Operand |
| 66 | DIVD2 | Divide Double 2 Operand |
| 67 | DIVD3 | Divide Double 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 such 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, the quotient operand is replaced by 0.
6. On floating divide by zero or on floating overflow the quotient operand is replaced by an operand of all bits 0 except for a sign bit of 1 (a reserved operand).

N ← 1; Z ← 0; V ← 1; and C ← 0.

**EXAMPLE:** DIVIDE FLOATING 2 OPERAND

```
DIVF2 R4, R2                    ;Divide
```

**Initial Conditions:**

R4 = 00004100
R2 = 00004330

**After Instruction Executions:**

R4 = 00004100
R2 = 000042B0

## EXTENDED DIVIDE

**Purpose:** perform extened-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:**

| | | |
|---|---|---|
| 78 | 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 such 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 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 operand is replaced by 0.

## BIT TEST

**Purpose:** perform arithmetic compare of one quantity to 0

**Format:** opcode mask.rx, src.rx

**Operation:** temp ← 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.

## 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 Long 2 Operand |
| C9 | BISL3 | Bit Set Long 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.

## 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 Long | 2 operand |
| CB | BICL3 | Bit Clear Long | 3 operand |

**Description:** In 2 operand format, the destination operand is ANDed with the ones 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 ones complement of the mask operand and the destination operand is replaced by the result.

## 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 Long 2 Operand |
| CD | XORL3 | Exclusive OR Long 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.

## 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 Long |
| 79 | ASHQ | Arithmetic Shift Quad |

**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 0 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. On overflow, the destination operand is replaced by the low order bits of the true result.
2. If cnt GTR 32 (ASHL) or cnt GRR 64 (ASHQ); the destination operand is replaced by 0.
3. If cnt LEQ —31 (ASHL) or cnt LEQ —63 (ASHQ); all the bits of the destination operand are copies of the sign bit of the source operand.

## ROTATE LONG

**Purpose:** rotate of 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 Long

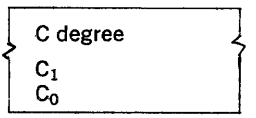
**Description:** The source operand is rotated logically by the number of bits specified by the count operand and the destination operand is replaced by the result. The source operand is unaffected. A positive count operand rotates to the left. A negative count operand rotates to the right. A 0 count operand replaces the destination operand with the source operand.

## POLYNOMIAL EVALUATION

**Purpose:** allows fast calculation of math functions

**Format:** opcode arg.rx, degree.rx, tbladdr.ab

**Operation:**




```
result ← C degree;

For degree times, loop
    result ← arg * result;
            !Perform multiply, and retain an
            !extended floating fraction of
            !31 bits (POLYF) or 63 bits (POLYD)
            !use this result in the following step
    result ← result + next coefficient;
            !normalize, round, and check for
            !over/underflow only after the
            !combined multiply/add sequence
    if overflow then trap;
    if underflow then clear result, remember
            underflow and continue looping;
```

**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 Floating |
| 75 | POLYD | Polynomial Evaluation Double |

**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.

The evaluation is carried out by Horner's method and the contents of R0 (R1'R0 for POLYD) are replaced by the result. The result computed is:

if d = degree
and x = arg
result = C[0] + x*(C[1] + x*(C[2] + . . . x*C[d]) )

The unsigned word degree operand specifies the highest numbered coefficient to participate in the evaluation.

**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

2. The multiplication is performed such that the precision of the product is equivalent to a floating point datum having a 31 bit (63 bit for POLYD) fraction.
3. If the unsigned word degree operand is 0, the result is C0.
4. If the unsigned word degree operand is greater than 31, a reserved operand exception occurs.
5. On a reserved operand exception:
   1. if PSL<FPD> = 0, the reserved operand is either the degree operand (greater than 31), or the argument operand, or some coefficient.
   2. if PSL<FPD> = 1, the reserved operand is a coefficient, and R3 is pointing at the value which caused the exception.
   3. 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 temporary result (tmp3) is replaced by zero, and the operation continues. A floating underflow trap occurs at the end of the instruction if underflow occurred during any iteration of the computation loop. Note that the final result may be non zero if underflow occurred before the last iteration.
7. On floating overflow after the rounding operation at any iteration of the computation loop, the instruction terminates and causes a trap. On overflow the contents of R2 and R3 (R2 through R5 for POLYD) are unpredictable. 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.

**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 7

# SPECIAL INSTRUCTIONS

This chapter describes instructions for manipulating the multiple registers, the processor status longword, addresses, indices, queues, and variable length bit fields. Most of these instructions represent optimizations of frequently occurring sequences of code.

Refer to Appendix E for a definition of the symbolic notation associated with the instruction descriptions.

## 7.1 MULTIPLE REGISTER INSTRUCTIONS

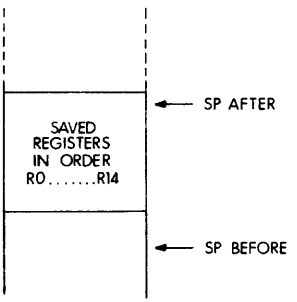
The multiple register instructions allow the saving and restoring of multiple registers in one operation. In both cases, the save area is on the stack. The PUSHR instruction saves multiple registers by pushing them onto the stack. The POPR instruction restores multiple registers by popping them from the stack. The list of registers is specified by a 16-bit mask operand with bit n representing register Rn. The mask operand is a normal read operand, so it can be calculated or 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 conventions for calling and signalling require that registers be saved in the call frame (see Appendix C and Chapter 8). Thus, any registers manipulated by PUSHR or POPR except R4 and R1 must appear in the procedure entry mask.

## PUSH REGISTERS

**Purpose:** save multiple registers or stack

**Format:** opcode mask.rw

**Operation:**




**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. 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 a double floating datum stored in adjacent registers being stored by PUSHR in memory in the correct order.

**EXAMPLES:**

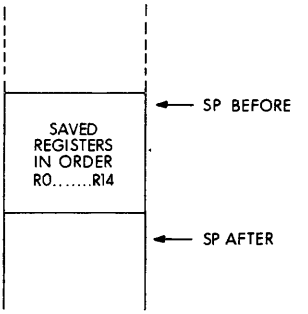
```
PUSHR #^M<R0, R1, R2, R3>      ;saves R0
                                through R3
PUSHR #^M<R1, R6, R7>          ;saves R1,
                                R6, and R7
```

## POP REGISTERS

**Purpose:** restore multiple registers from stack

**Format:** opcode mask.rw

**Operation:**




**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. Bit 15 is ignored.

# 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

**Opcodes:**

| | | |
|---|---|---|
| DC | MOVPSL | Move from PSL |

**Description:** The destination operand is replaced by the processor status longword (see Chapter 12).

## 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 ones 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
```

# MOVA
# PUSHA

## PUSH ADDRESS
## MOVE 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 ← result LSS 0;
Z ← result EQL 0;
V ← 0;
C ← C
```

**Exceptions:** None

**Opcodes:**

| | | |
|---|---|---|
| 9E | MOVAB | Move Address Byte |
| 3E | MOVAW | Move Address Word |
| DE | MOVAL | Move Address Long |
| DE | MOVAF | Move Address Floating |
| 7E | MOVAQ | Move Address Quad |
| 7E | MOVAD | Move Address Double |
| 9F | PUSHAB | Push Address Byte |
| 3F | PUSHAW | Push Address Word |
| DF | PUSHAL | Push Address Long |
| DF | PUSHAF | Push Address Floating |
| 7F | PUSHAQ | Push Address Quad |
| 7F | PUSHAD | Push Address Double |

**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.

**EXAMPLE:**
```
PUSHA XYZ        ;pushes the address of XYZ
```

## 7.4 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 allows index calculation optimization by removing invariant expressions.

## 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 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.

**EXAMPLES:** 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/1 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:

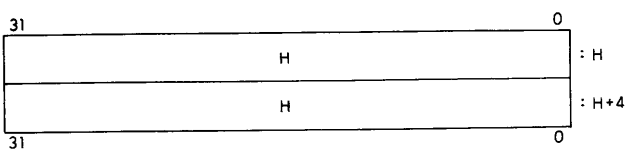
```
INDEX J, #L2, #U2, #M1, #0, R0;
  M1=U1-L1+1
INDEX I, #L1, #U1, #1, R0, R0;
MOVL #1, A-a[R0]; a = ((L2*M1) + L1) *4
```

## 7.5 QUEUE INSTRUCTIONS

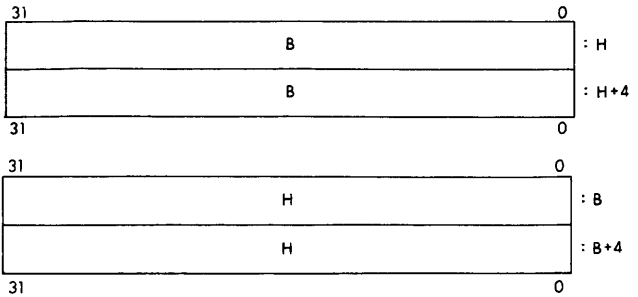
A queue is a circular, doubly linked list. Each queue entry is linked to the next via a pair of longwords. The first (lowest addressed) longword is the forward link: the address of the succeeding queue entry. The second (highest addressed) longword is the backward link: the address of the preceding queue entry. A 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. (Under certain restrictions they can be inserted or removed elsewhere; this is discussed later.)

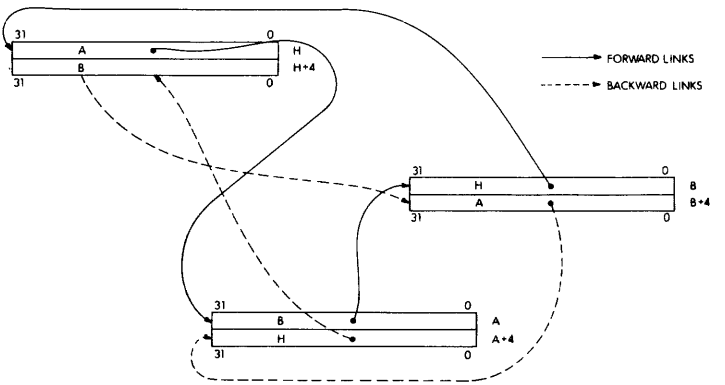
The following contains examples of queue operations. An empty queue is specified by its header at address H:



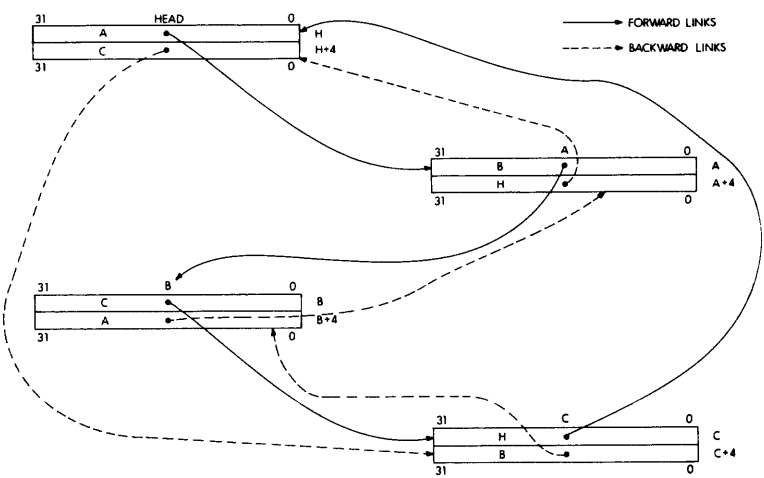

Insertion into an empty queue (at either the head or tail) of an entry at address B gives:




Insertion at the head of an entry at address A gives:

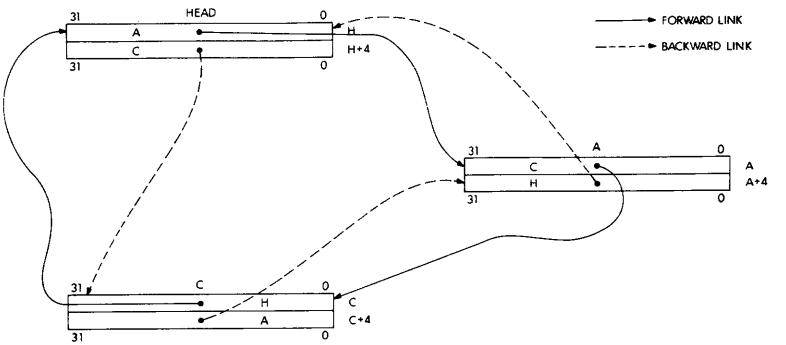



Finally insertion at the tail of entry at address C gives:



Following the above steps in reverse order gives the effect of removal at the tail and removal at the head.

If more than 1 process can perform operations on a queue simultaneously, insertions and removals should only be done at the head or tail of the queue. If only 1 process (or 1 process at a time) can perform operations on a queue, insertions and removals can be made at other than the head or tail of the queue. In the example above 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 simultaneously performing operations on the queue.

## 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
{interrupts off per notes 1 and 2};
(entry+4) ← pred;                !forward link of entry
(pred+4) ← entry;                !backward link of entry
(pred) ← entry;                  !backward link of
                                  successor
(pred) <— entry;                 !forward link of
                                  predecessor
{interrupts on};
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:** reserved operand

**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 non-interruptible interlocked. 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 non-interruptible, processes running in kernel mode can share queues with interrupt service routines.
2. The INSQUE and REMQUE instructions are implemented such that cooperating software processes may access a shared list without additional synchronization.
3. During access validation, any access which cannot be completed results in a memory management exception even though the queue insertion is not started.
4. A reserved operand fault occurs if any of entry, pred, or (pred) is an address that is not longword aligned (i.e., for which <1:0> NEQU 0). In this case, the queue is not altered.

**EXAMPLES:**

1. Insert at head
```
INSQUE      entry,h          ;h is queue head
```
2. Insert at tail
```
INSQUE      entry,@h+4       ;h is queue head
```
(Note "@" in this case only)

3. 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
```

## 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
{interrupts off per notes 1 and 2};
( (entry+4) ) ← (entry); !forward link of predecessor
( (entry)+4) ← (entry +4); !backward link of successor
addr ← entry;
{interrupts on};
end;
```

**Condition Codes:**

```
N ← (entry) LSS (entry+4);
Z ← (entry) EQL (entry+4);      !removed last entry
V ← entry EQL (entry+4);        !no entry to remove
C ← (entry) LSSU (entry+4);
```

**Exceptions:** reserved operand

**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 entry removed was the last entry in the queue, the condition code Z-bit is set; otherwise it is cleared. The removal is a non-interruptible 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 non-interruptible, processes running in kernel mode can share queues with interrupt service routines.
2. The INSQUE and REMQUE instructions are implemented such that cooperating software processes 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.
4. A reserved operand fault occurs if any of entry (entry), or (entry+4) is an address that is not longword aligned (i.e., for which <1:0> NEQU 0). In this case, the queue is not altered.

**EXAMPLES:**

1. Remove at head
```
REMQUE      @h,addr        ;h is queue header
```
2. Remove at tail
```
REMQUE      @h+4, add2     ;h is queue header
```
3. 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$                    ;
```

## 7.6 VARIABLE LENGTH BIT FIELD INSTRUCTIONS

The variable length bit field instructions are useful when dealing with data not in 8-bit increments (for example, 13 bits of data not starting on a byte boundary). This data could also be handled without this group of instructions but it would require additional shift and mask operations to get the bits in the proper form and to eliminate the non-required bits.

A variable bit field is 0 to 32 contiguous bits that may be contained in 1 to 5 bytes and 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 specifiers are:

1. Position operand—a signed longword operand that designates the number of bits away from the base address operand.

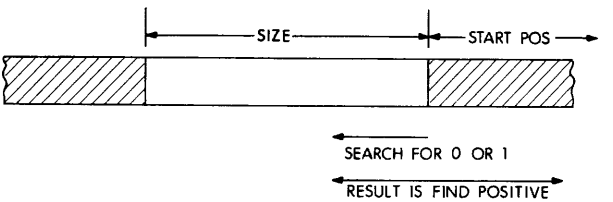
   If the variable length field is contained in a register, the position operand must have a value in the range 0 through 31 or a reserved operand fault occurs.
2. Size Operand—a byte operand which specifies the length of the field. This operand must be in the range 1 through 32 or a reserved operand fault occurs. The size operand will normally be a short literal if the field is fixed.
3. Base Address—an address relative to the position 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 register n+1 concatenated with register n).

## FIND FIRST

**Purpose:** locate first bit in bit field

**Format:** opcode startpos.rl, size.rb, base.ab, 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 a 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.

The Find First instruction is 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 and 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.

**Note:** If start position + size is GEQU 2**31, then find position

might be set to a negative value that would not be usable in a subsequent field or BBxx instruction.

**EXAMPLE:**

```
FIND FIRST SET
FFS #5, #10, 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, Work 1, R2        ;Find first clear bit
                               ;in Work1
```

**Initial Conditions:**

Work1 = ^XF0
R2 = 00000000

**After Instruction Execution:**

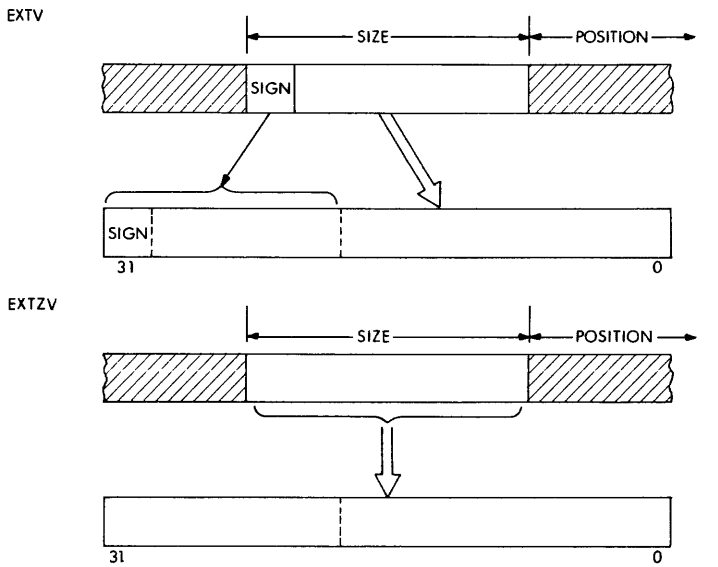
Work1 = ^XF0
R2 = 00000008

## EXTRACT FIELD

**Purpose:** moves bit field to integer

**Format:** opcode pos.rl, size.rb, base.vb, dst.wl

**Operation:**




**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 as opposed to the extract Field instruction.

**Notes:**

1. A reserved operand fault occurs if:
   a. size GTRU 32
   b. pos GTRU 31 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:**

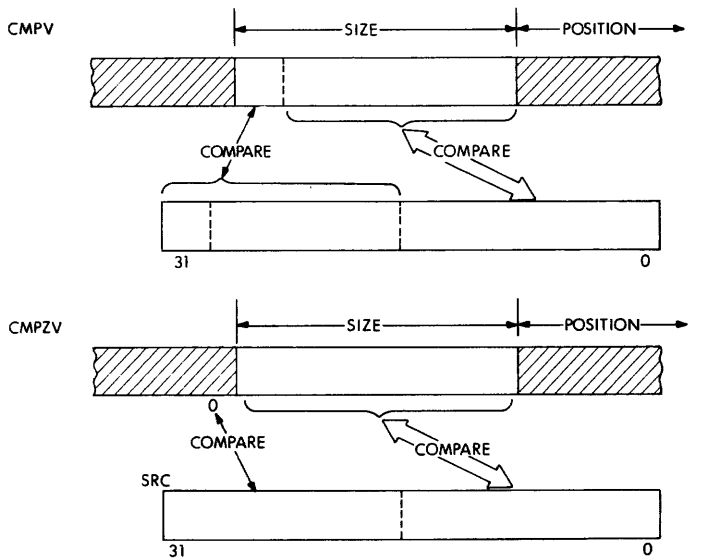
```
Work1 = 00004F04
R1 = 00000478
```

## COMPARE FIELD

**Purpose:** compare bit field to integer

**Format:** opcode pos.rl, size.rb, base.ab, src.rl

**Operation:**




**Codition Codes:**
N ← field LSS src;
Z ← field EQL src;
V ← 0;
C ← field 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:**

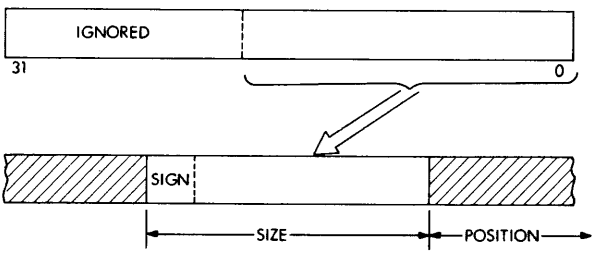
1. A reserved operand fault occurs if:
   a. size GTRU 32
   b. pos GTRU 31 and the field is contained in the registers.
2. On a reserved operand fault, the condition codes are unpredictable.
3. The comparison is with the entire source operand longword, not just the size field.

## INSERT FIELD

**Purpose:** move integer to bit field

**Format:** opcode src.rl, pos.rl, size.rb, base.ab

**Operation:**




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

**Exceptions:** Reserved operand

**Opcodes:** FO 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 contion codes.

**Notes:**
1. A reserved operand fault occurs if:
   a. size GTRU 32
   b. pos GTRU 31 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

# CHAPTER 8

# CONTROL INSTRUCTIONS

This chapter describes the branch, loop, control, subroutine, case, and call classes of instructions. In most implementations of the VAV-11 architecture, improved execution speed will result if the target of a control instruction is on an aligned longword boundary.

Refer to Appendix E for a definition of the symbolic notation associated with the instruction descriptions.

## 8.1. 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. 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 instruction, and branch instructions are more efficient than jump instructions, the processor offers a variety of branch instructions to choose from. There are two unconditional branch instructions (branch and jump) and many conditional branch instructions.

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 away from the current location as 32,767 bytes in either direction. For control transfers to locations farther away, you must use the Jump instruction (JMP).

Two special types of branch and jump instruction are provided for calling subroutines: the Branch to Subroutine (BSB) and Jump to Subroutine (JSB) instructions. 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 short-cut 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.

## 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-11 conditional branch instructions permit considerable flexibility in branching but require care in choos-

ing the correct branch instruction. The conditional branch instructions are divided into 3 overlapping groups:

1. 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. 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, address instructions, and character string instructions.

3. 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.

## 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

**Exception:** 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.

## BRANCH ON BIT

**Purpose:** test selected bit

**Format:** opcode pos.rl, base.ab, 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 INTERLOCK)

**Purpose:** test and modify selected bit

**Format:** opcode pos.rl, base.ab, 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.ab, displ.bb

**Operation:**
```
teststate = if {BBSSI} then 1 else 0;
newstate = teststate;
{set interlock};
tmp ← FIELD (pos, 1, base);
FIELD (pos, 1, base) ← newstate;
{release interlock};
if temp 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 is an interlocked operation. No other processor or I/O device can do an interlocked access on the bit during the interlocked operation.

**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 modify interlocks with other processors or devices. For example, to implement "busy waiting":

```
1$:    BBSSI    bit,base,1$
```

## BRANCH ON LOW BIT

**Purpose:** test bit

**Format:** opcode src.rl, displ.bb

**Operation:**
```
teststate = if (BLS) 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.

**Note:** The source operand is taken with longword context although only one bit is tested.

## 8.2 LOOP CONTROL INSTRUCTIONS

## 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 Long |
| 4F | ACBF | Add Compare and Branch Floating |
| 6F | ACBD | Add Compare and Branch Double |

**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, 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, the index operand is replaced by 0. Comparison and branch determination proceed normally.
4. On floating overflow, the index operand is replaced by an operand of all bits 0 except for a sign bit of 1 (a

reserved operand). N ← 1; Z ← 0; V ← 1. The branch is not taken.

5. On a reserved operand fault, the index operand is unaffected and the condition codes are unpredictable.
6. Except for 5. above, the C-bit is unaffected.
7. On a trap, the branch condition will be tested and the PC potentially updated before the exception is taken. Thus, the PC might point to the start of the loop and not the next consecutive instruction.

## ADD ONE AND BRANCH

**Purpose:** increment integer loop count and loop

**Format:** opcode limit.rl, index.ml, displ.bb

**Operation:**

```
index ← index +1;                          !AOBLSS
if index LSS limit
then PC ← PC + SEXT (displ);
if index LEQ limit                         !AOBLEQ
then PC + SEXT (displ);
```

**Condition Codes:**

```
N ← index LSS 0;
Z ← index EQL 0;
V ← {integer overflow};
C ← C
```

**Exception:** 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) the branch is taken.
2. The C-bit is unaffected.

## SUBTRACT ONE AND BRANCH

**Purpose:** decrement integer loop count and loop

**Format:** opcode index.ml, displ.bb

**Operation:**

```
index ← index −1;                          !SOBGEQ
if index GEQ 0 then PC ←
PC + SEXT (displ);
If index GTR 0 then PC ←                   !SOBGTR
PC + SEXT (displ);
```

**Condition Codes:**

```
N ← index LSS 0;
Z ← index EQL 0;
V ← {integer overflow};
C ← C
```

**Exception:** 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.

## 8.3 CASE INSTRUCTIONS

### CASE

**Purpose:** perform multi-way 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 temp LEQU limit then
SEXT (displ [tmp]) else {2 + 2 * ZEXT (limit)};

**Condition Codes:**
N ← temp LSS limit;
Z ← temp EQL limit;
V ← 0;
C ← tmp LSSU limit

**Exceptions:** none

**Opcodes:**

| | | |
|---|---|---|
| 8F | CASEB | Case Byte |
| AF | CASEW | Case Word |
| CF | CASEL | Case Long |

**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 and 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 higher-level language computed GO TO statements: the CASE instruction. 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$         ;if 1
     .WORD 20.—1$         ;if 2
     .WORD 30.—1$         ;if 3
                          ;fall through if out of range
```

## 8.4 SUBROUTINE INSTRUCTIONS

## JUMP, BRANCH 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 1 byte shorter.

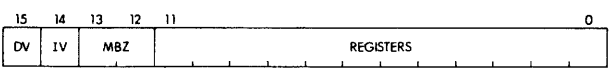
## 8.5 PROCEDURE CALL INSTRUCTIONS

Procedures are general purpose routines that use argument lists passed automatically by the processor and use only local variables for data storage. The Procedure Call instructions provide several services. They:

- save all the registers that the procedure uses, and only those registers, before entering the procedure
- pass an argument list to a procedure
- maintain the Stack, Frame, and Argument Pointer registers
- set 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. Refer to Appendix C for the procedure calling standard. The CALLG instruction calls a procedure with the argument list actuals 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. The entry point 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.

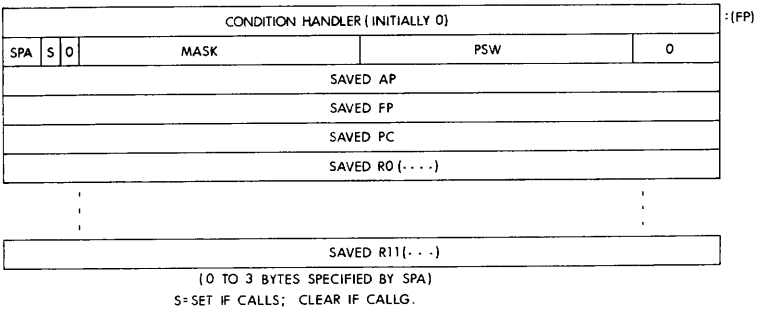
The entry mask specifies the subprocedure's register use and overflow enables:




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.

The registers 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 as 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 can be considered as 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. The stack frame has the following format:




Note that the saved condition codes 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. Similarly, the saved trace enable (PSW<T>) is cleared.

The software defines symbolic names for the fixed fields in the call frame as follows:

| Mnemonic | Value | Meaning |
|---|---|---|
| SRM$A_HANDLER | 0 | condition handler |
| SRM$W_SAVE_PSW | 4 | saved PSW |
| SRM$W_SAVE_MASK | 6 | SPA'S '0' mask |
| SRM$L_SAVE_AP | 8 | saved AP |
| SRM$L_SAVE_FP | 12 | saved FP (backward link) |
| SRM$L_SAVE_PC | 16 | saved PC |
| SRM$L_SAVE_REGS | 20 | start of saved R0 . . . . R11 |

The save_mask fields have symbolic names as follows:

| Mnemonic | Value | Meaning |
|---|---|---|
| SRM$V_REGMASK | 0 | position of register mask |
| SRM$_REGMASK | 12 | size of register mask |
| SRM$V_CALLS | 13 | CALLS flag |
| SRM$V_STACKOFFS | 14 | position of stack alignment |
| SRM$$_STACKOFFS | 2 | size of stack alignment |

## 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}

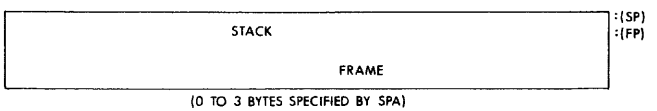
**Condition Codes:**
N ← 0;
Z ← 0;
V ← 0;
C ← 0

**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 0 and the contents of registers whose number corresponds to set bits in the mask are pushed on the stack as longwords. 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 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. 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 plus 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, condition 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 R2 through R11 which are modified in the called procedure must be preserved in the mask. Refer to Appendix C.

## 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}

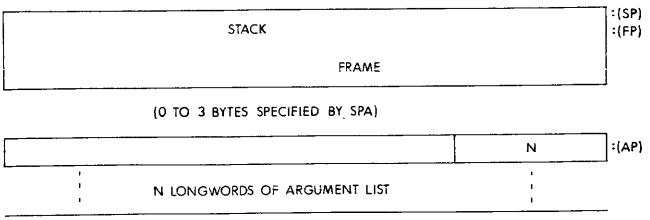
**Condition Codes:**
N ← 0;
Z ← 0;
V ← 0;
C ← 0

**Exceptions:** reserved operand

**Opcodes:**

| | | |
|---|---|---|
| FB | CALLS | Call Procedure With Stack Argument List |

**Description:** The number of arguments operand is pushed on the stack as a longword. 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 registers 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. T-bit is unaffected. AP is replaced by the saved SP. PC is replaced by the sum of destination operand plus 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:




**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 R2 through R11 which are modified in the called procedure must be preserved in the entry mask. Refer to Appendix C.

## RETURN FROM PROCEDURE

**Purpose:** transfer control from a procedure back to calling program

**Format:** opcode

**Operation:**
```
{restores SP from FP};
{restore registers};
{drop stack alignment};
{restore PSW};
{If CALL$, remove arglist}
```

**Condition Codes:**
```
N ← restored PSW<3>;
Z ← restored PSW<2>;
V ← restored PSW<1>;
C ← restored PSW<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, CF, 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 replaced by the sum of SP and 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 tmpl<15:8> NEQ 0.
2. On a reserved operand fault, the condition codes are unpredictable. The value of tmpl<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. See Appendix C.

# CHAPTER 9

# CHARACTER STRING INSTRUCTIONS

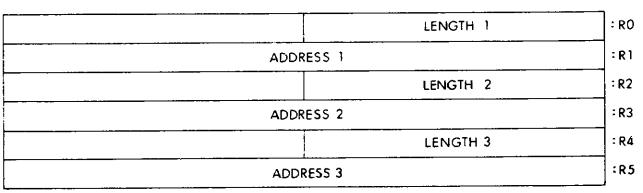
This chapter describes the character string instructions and the CRC (Cyclic Redundancy Check) instruction.

## 9.1 CHARACTER STRING INSTRUCTIONS

A character string is specified by 2 operands:

1. An unsigned word operand which specifies 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 through R1, R0 through R3, or R0 through R5 to contain a control block which maintains updated addresses and state during the execution of the instruction. At completion, these registers are available to software to use as string specification operands for a subsequent instruction on a contiguous character string. During the execution of the instructions, pending interrupt conditions are tested and if any is found, the control block is updated, a first part done bit is set in the PSL, and the instruction interrupted (see Chapter 12). After the interruption, the instruction resumes transparently. The format of the control block is:




The fields LENGTH 1, LENGTH 2 (if required) 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 (if required) and ADDRESS 3 (if required) contain the address of the next byte to be processed in the first, second, and third string operands respectively.

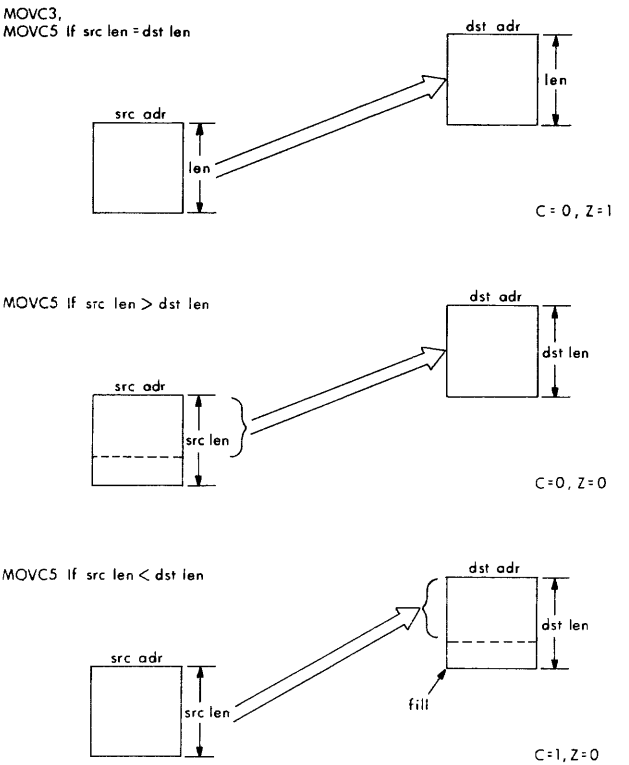
Refer to Appendix E for a description of the symbolic notation associated with the instruction descriptions.

## MOVE CHARACTER

**Purpose:** to move character string or block of memory

**Format:**

| | |
|---|---|
| opcode len.rw, srcaddr.ab, dstadr.ab | 3 operand |
| opcode srclen.rw, srcaddr.ab, fill.rb, dstlen.rw, distaddr.ab | 5 operand |




**Condition Codes:**

N ← srclen LSS dstlen;
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:** In 3 operand format, 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. In 5 operand format, the destination string specified by the destination length and destination address operands is replaced by the source string specified by the source length and source address operands. If the destination string is longer than the source string, the highest addressed bytes of the destination 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 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 non-zero 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

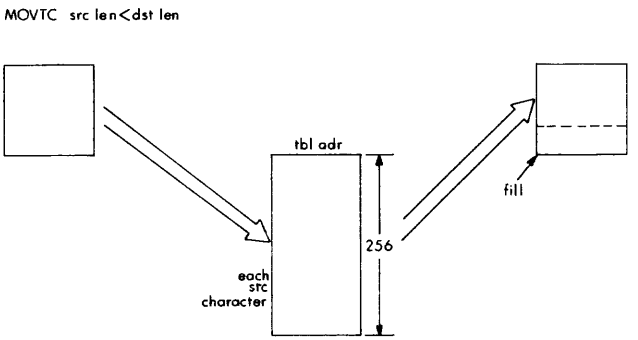
3. MOVC3 is the preferred way to copy one block of memory to another.
4. MOVC5 with a 0 source length operand is the preferred way to fill a block of memory with the fill character.

## 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:**




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 specified by the source length and source address operands is translated and replaces the destination string specified by the destination length and destination address operands. 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 non-zero 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

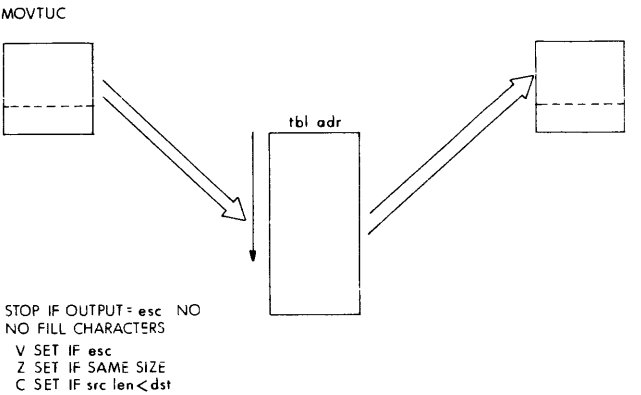
R5 = address of one byte beyond the destination string

## 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 by the source length and source address operands is translated and replaces the destination string specified by the destination length and destination address operands. Translation is accomplished by using each byte of the source string as 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 source string or the table, the destination string is unpredictable.

**Notes:** After execution:
R0 = number of bytes remaining in source string (in-

cluding 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.

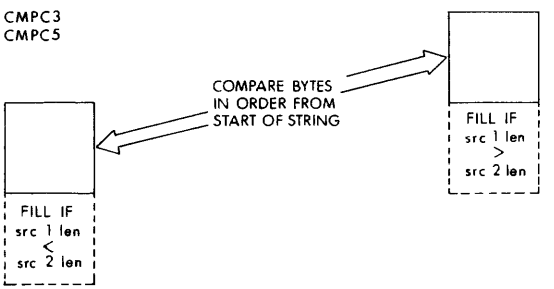
## COMPARE CHARACTERS

**Purpose:** to compare two character strings

**Format:**

| | |
|---|---|
| opcode len.rw, src1addr.ab, src2addr.ab | 3 operand |
| opcode src1len.rw, src1addr.ab, fill.rb, src2len.rw, src2addr.ab | 5 operand |

**Operation:**




NOTE: CONDITION CODES SET ON LAST COMPARE DONE

**Condition Codes:**
N ← (string 1 terminal byte) LSS (string 2 terminal byte);
Z ← (string 1 terminal byte) EQL (string 2 terminal byte);
!strings are equal
V ← 0;
C ← (string 1 terminal byte) LSSU (string 2 terminal 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 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. Com-

parison 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 = adress 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.

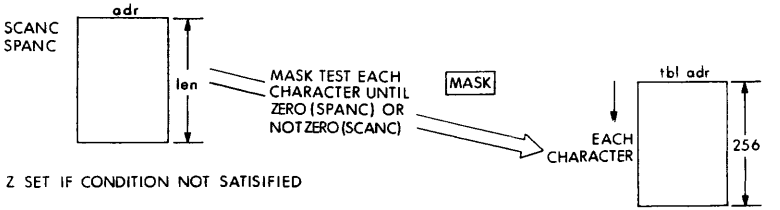
   R3 = address of the byte in string 2 which terminated comparison; if comparison did not terminate before string 2 was exhausted, R3 = address of one byte beyond string 2.

## SCAN CHARACTERS, SPAN CHARACTERS

**Purpose:** to find or skip a set of characters in character string

**Format:** opcode len.rw, addr.ab, tbladdr.ab, mas.rb

**Operation:**




**Condition Code:**
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 non-zero for the SCANC instruction or zero for the SPANC instruction, or until all the bytes of the string have been exhausted. If a non-zero 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:** After execution:

R0 = number of bytes remaining in the string (including the byte which produced the non-zero AND result for SCANC or zero result for SPANC)

R0 is zero only if there was a zero AND result for SCANC or a non-zero result for SPANC.

R1 = address of the byte which produced non-zero AND result for SCANC or a zero AND result for SPANC; or, if zero result, R1 = address of one byte beyond the string

R2 = 0

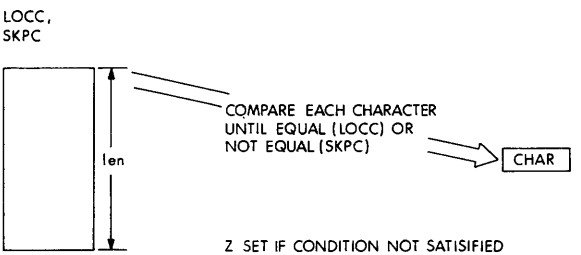
R3 = address of the table

## 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 | SKP | 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:** After execution:

R0 = number of bytes remaining in the string (including located one) if byte located; otherwise R0 = 0.

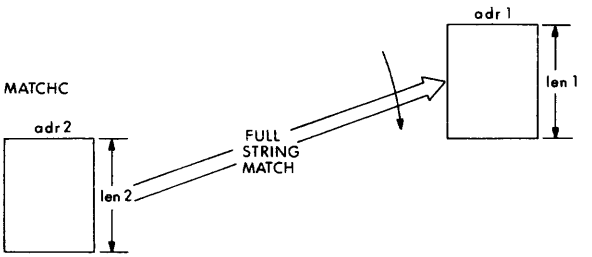
R1 = address of the byte located if byte located; otherwise R1 = address of one byte beyond the string.

## MATCH CHARACTERS

**Purpose:** to find substring in character string

**Format:** opcode len1.rw, addr1.ab, len2.rw, addr2.ab

**Operation:**




**Condition Codes:**
N ← 0;
Z ← R0 EQL 0;
V ← 0;
C ← 0

**Exceptions:** None

**Opcodes:** 39 MATCHC Match Characters

**Description:** The string specified by the length 1 and address 1 operands is searched for a substring which matches the string specified by the length 2 and address 2 operands. If the substring is found, the condition code Z-bit is cleared; otherwise, it is set.

**Notes:** After execution:

R0 = number of bytes remaining in string 1 including bytes of the matched substring. R0 is 0 only if no match occurred.

R1 = address of the first byte of the substring if substring match occurred; otherwise, address of one byte beyond string 1.

R2 = number of bytes in string 2.

R3 = address of the first byte of string 2.

## 9.2 CYCLIC REDUNDANCY CHECK INSTRUCTION

This instruction is designed to implement the calculation and checking of a cyclic redundancy check for any CRC polynomial up to 32 bits. Cyclic Redundancy Checking 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-11 string in memory. Error detection is accomplished by computing the CRC at the source and again at the destination, comparing the CRC computed at each end. The choice of the polynomial is such as to minimize the number of undetected block errors of specific lengths. The choice of a CRC polynomial is not given here; see, for example, the article "Cylic Codes for Error Detection" by W. Peterson and D. Brown in the Proceedings of the IRE (January, 1961).

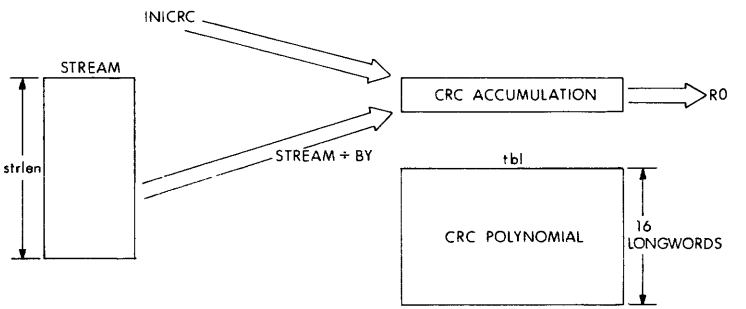
The operands to the CRC instruction are a string descriptor, a 16-longword table, and an initial CRC. The string descriptor is a standard VAX-11 operand pair of the length of the string in bytes (up to 65,535) and the starting address of the string. The contents of the table are a function of the CRC polynomial to be used. It can be calculated from the polynomial by the algorithm in the notes. Several common CRC polynomials are also included in the notes. 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 is 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 8 bits of the CRC. Then the CRC is shifted right 1 bit, inserting zero on the left. The right most 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. The actual algorithm 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. The data stream must be a multiple of eight bits in length. If it is not, the stream must be right adjusted in the string with leading 0 bits.

## CALCULATE CYCLIC REDUNDANCY CHECK

**Format:** opcode tbl.ab, inicrc.rl, strlen.rw, stream.ab, dst.wl

**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 8-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 = ISHFT (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:

| shift | loop | test | table increment | use table 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.
7. After execution:
   R0 = resultant CRC
   R1 = address of one byte beyond the stream
   R2 = 0
   R3 = 0

# CHAPTER 10

# DECIMAL STRING INSTRUCTIONS

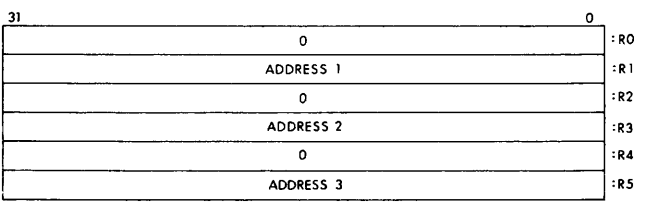
Decimal string instructions operate on packed decimal strings. Convert instructions are provided between Packed Decimal and Trailing Numeric String (Overpunched and Zoned) and Leading Separate Numeric string formats. Where necessary, a specific data type is identified. Where the phrase "decimal string" is used, it means any of the three data types.

A decimal string is specified by 2 operands:

1. For all 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 Left Separate Numeric strings. The address is specified by a byte operand of address access type.

Each of the decimal string instructions uses general registers R0 through R3 or R0 through R5 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 is found, the control block is updated. First Part Done is set in the PSL, and the instruction interrupted. After the interruption, the instruction resumes transparently. The format of the control block at completion is:




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.

The decimal string instructions 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.

## 10.1 DECIMAL OVERFLOW

Decimal overflow occurs if the destination string is too short to contain all the non-zero 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.

## 10.2 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 compares equal to −0. 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.

## 10.3 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.

## 10.4 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. The destination strings, registers used by the instruction, and condition codes will, in general, be unpredictable when a reserved operand fault occurs.

## 10.5 PACKED DECIMAL OPERATIONS

Packed decimal strings generated by the decimal string instructions always have the preferred sign representation: 12 for "+" and 13 for "−". 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:

1. A digit occurs in the sign position.
2. A sign occurs in a digit position.
3. For an even length string, a non-zero nibble occurs in the high order nibble of the lowest addressed byte.

## 10.6 ZERO LENGTH DECIMAL STRINGS

The length of a packed decimal string can be 0. 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 0. 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 0. 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 0.

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

## MOVE PACKED

**Format:** opcode len.rw, srcaddr.ab, dstaddr.ab

**Operation:** {dst} ← {src string}

**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 destination string, R0 through R3, and the condition codes are unpredictable if the destination string overlaps the source string, the source string contain 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.

## COMPARE PACKED

**Format:**

| | |
|---|---|
| opcode len.rw, src1addr.ab, src2addr.ab | 3 operand |
| opcode src1len.rw, src1addr.ab, src2len.rw, rsc2addr.ab | 4 operand |

**Operation:** {src1 string} − {src2 string};

**Condition Codes:**
N ← {src1 string} LSS {src2 string};
N ← {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.

## ADD PACKED

**Format:** opcode addlen.rw, addaddr.ab, sumlen.rw, sumaddr.ab

opcode add1len.rw, add1addr.ab, add2len.rw, add2addr.ab, sumlen.rw, sumaddr.ab

**Operation:** {sum string} ← {sum string} + {add string}; !4 operand
{sum string} ← {add 1 string} + {add2 string}; !6 operand

**Condition Codes:**
N ← {sum string} LSS 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 aded 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:

   R = 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 fault occurs.
4. If all destination digits are zero, Z is set and N is cleared. This is true even if the result overflows.

## SUBTRACT PACKED

**Format:**

```
opcode sublen.rw, subaddr.ab, diflen.rw,          4 operand
difaddr.ab

opcode sublen.rw, subaddr.ab, minlen.rw,          6 operand
minaddr.ab, diflen.rw, difaddr.ab
```

**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 fault occurs.
4. If all destination digits are zero, Z is set and N is cleared. This is true even if the result overflows.

## MULTIPLY PACKED

**Format:** opcode mulrlen.rw, mulraddr.ab, muldlen.rw, muladdr.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 fault occurs.
3. If all destination digits are zero, Z is set and N is cleared. This is true even if the result overflows.

## DIVIDE PACKED

**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:
   1. The absolute value of the remainder (which is lost) is less than the absolute value of the divisor.
   2. The product of the absolute value of the quotient times the absolute value of the divisor is less than or equal to the absolute value of the dividend.
   3. 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 fault occurs.
5. If all destination digits are zero, Z is set and N is cleared. This is true even if the result overflows.

## CONVERT LONG TO PACKED

**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 fault.
3. If the destination digits are zero, Z is set and N is cleared. This is true even if the result overflows.
4. Overlapping operands produce correct results.

## CONVERT PACKED TO LONG

**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. If the source string has a value outside the range −2,147,483,648 through 2,147,483,647, integer overflow occurs 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.

## CONVERT PACKED TO TRAILING NUMERIC

**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 fault 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 non-zero 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 negated conversions.
5. 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).

## CONVERT TRAILING NUMERIC TO PACKED

**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 4 bits of the corresponding bytes in the source string.

**Notes:**

1. A reserved operand fault occurs if:
   1. The length of the source trailing numeric string is outside the range 0 through 31.
   2. The length of the destination packed decimal string is outside the range 0 through 31.
   3. 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).
   4. 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.

## CONVERT PACKED TO LEADING SEPARATE NUMERIC

**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 with 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 fault 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.

## CONVERT LEADING SEPARATE NUMERIC TO PACKED

**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 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:
   1. The length of the source Leading Separate numeric string is outside the range 0 through 31.
   2. The length of the destination packed decimal string is outside the range 0 through 31.
   3. 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 fault occurs.

## ARITHMETIC SHIFT AND ROUND PACKED

**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 fault 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 non-zero, 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.

# CHAPTER 11

# EDIT INSTRUCTION

This instruction is designed to implement the common editing functions which occur in handling fixed format output. It operates by converting a packed decimal string to a character string. This operation is exemplified by a MOVE to a numeric edited (PICTURE )item in COBOL or PL/I, but the instruction can be used for other applications as well. The operation consists of converting an input packed decimal number to an output character string, generating characters for the output. When converting digits, options include leading zero fill, leading zero protection, insertion of floating sign, insertion of floating currency symbol, insertion 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 is a standard VAX-11 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 much the way that the normal instructions are. The output string is described by only its starting address 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 asterisk for check protection. The other character register contains the sign character. Initially this contains either an ASCII blank or a minus sign depending upon the sign of the input. This can be changed to allow other sign representations such as plus/minus 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 contain 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 R0-R5 contain the conventional values after a decimal instruction.

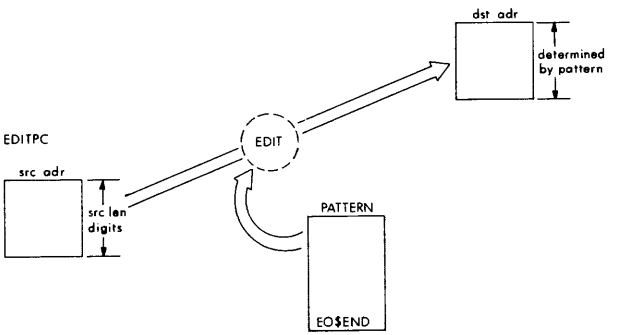
Refer to Appendix E for a description of the symbolic notation associated with the instruction descriptions.

## EDIT PACKED TO CHARACTER STRING

**Purpose:**

**Format:** opcode srclen.rw, scraddr.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}; !non-zero 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 operands is replaced by the edited version of the source string specified by the source length and source address operands. The 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 the following pages.

**Notes:**

1. A reserved operand fault occurs with FPD cleared if srclen GTRU 31. See Chapter 6 for a description of reserved operand faults and FPD.

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$BLANK_ZERO of EO$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 = non-zero digits lost

    C = significance

    R0<15:0> = srolen

    R1<31:16> = — number of zeros to supply

    R1 = current source location

    R2<7:0> = fill character

    R2<15:8> = sign character

    R2<31:16> = unpredictable

    R3 = address of edit pattern operator causing exception

    R4 = unpredictable

    R5 = location of next destination byte

**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 count 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$STORE_SIGN | |
| 05 . . 1F | Reserved to DEC | |
| 20 . . 3F | Reserved for all time | |
| 40 | EO$LOAD_FILL | character is in next byte |
| 41 | EO$LOAD_SIGN | |
| 42 | EO$LOAD_PLUS | |
| 43 | EO$LOAD_MINUS | |
| 44 | EO$INSERT | |
| 45 | EO$BLANK_ZERO | unsigned length is in next byte |
| 46 | EO$REPLACE_SIGN | |
| 47 | EO$ADJUST_INPUT | |
| 48 . . 5F | Reserved to DEC | |
| 60 . . 7F | Reserved to CSS, customers | |
| 80, 90, A0 | Reserved to DEC | |
| 81 . . 8F | EO$FILL | repeat count is <3:0> |
| 91 . . 9F | EO$MOVE | |
| A1 . . AF | EO$FLOAT | |
| B0 . . FE | Reserved to DEC | |
| FF | Reserved for all time | |

The following pages define each pattern operator in a format similar to that of the normal instruction descriptions. In each case, if there is an operand it is either a repeat count (r) from 1 through 15, an unsigned byte length (len), or a character byte (ch). In the formal descriptions, the following two routines are invoked:

```
READ:                   !function value 0 through 9
        if R0 LEQ 0
        then
                        begin
                        if R0 EQL 0 then {reserved operand};
                        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>

## INSERT CHARACTER

**Purpose:** insert a fixed character, substituting the fill character if not significant

**Format:** pattern ch

**Operation:** 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 significance is set, then the character is placed into the destination. If significance is not set, then the contents of the fill register is placed into the destination.

**Note:** 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).

## STORE SIGN

**Purpose:** Insert the sign character

**Format:** pattern

**Operation:** STORE (sign);

**Pattern Operators:** 04 EO$STORE_SIGN Store Sign

**Description:** The contents of the sign register is placed into the destination.

**Note:** This pattern operator is used for any non-floating arithmetic sign. It should be preceded by a EO$LOAD_PLUS and/or EO$LOAD_MINUS if the default sign convention is not desired.

## STORE FILL

**Purpose:** Insert the fill character

**Format:** pattern r

**Operation:** 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 is placed into the destination repeat times.

**Note:** This pattern operator is used for fill (blank) insertion.

## MOVE DIGITS

**Purpose:** Move digits, filling for insignificant digits (leading zeros)

**Format:** pattern r

**Operation:**

```
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. If the digit is non-zero, significance is set and zero is cleared. 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 zeros should be explicit, significance 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.

## FLOAT SIGN

**Purpose:** Move digits, floating the sign across insignificant digits

**Format:** pattern r

**Operation:**

```
repeat r do
        begin
        tmp ← READ;
        if tmp NEQU 0 then
           begin
           if PSW<C> EQL 0 then STORE (sign);
           PSW<Z> ← 0;
           PSW<C> ← 1;        !set significance
           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. If it is non-zero and significance is not yet set, then the contents of the sign register is stored in the destination, significance is set, and zero is cleared. If the digit is significant, it is stored in the destination, otherwise the contents 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 setup 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$END_FLOAT.
3. This pattern operator is used to move digits with a floating currency sign. The sign must already be setup with a EO$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.

## END FLOATING SIGN

**Purpose:** End a floating sign operation

**Format:** pattern

**Operation:**

```
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 is stored in the destination and significance is set.

**Note:** 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.

## BLANK BACKWARDS WHEN ZERO

**Purpose:** Fixup the destination to be blank when the value is zero

**Format:** pattern len

**Operation:**

```
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 is stored into the last length bytes of the destination string.

**Notes:**

1. The length must be non-zero 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.

## REPLACE SIGN WHEN MINUS ZERO

**Purpose:** Fixup the destination sign when the value is minus zero

**Format:** pattern len

**Operation:**
```
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 minus zero (i.e., if both N and Z are set), then the contents of fill register is stored into the byte of the destination string length before the current position.

**Notes:**

1. The length must be non-zero and within the destination string already produced. If it is not, the contents of the destination string and the memory preceding it are inpredictable.
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.

## LOAD REGISTER

**Purpose:** Change the contents of the fill or sign register

**Format:** pattern ch

**Operation:**

```
                                !select one depending on pattern operator
fill ← ch;                      !EO$LOAD_FILL
sign ← ch;                      !EO$LOAD_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. For EO$LOAD_PLUS this character is placed into the sign register if the source string has a positive sign. For EO$LOAD_MINUS this character is placed into the sign register if the source string has a negative sign.

**Notes:**

1. EO$LOAD_FILL is used to setup check protection (* instead of space).
2. EO$LOAD_SIGN is used to setup a floating currency sign.
3. EO$LOAD_PLUS is used to setup a non-blank plus sign.
4. EO$LOAD_MINUS is used to setup a non-minus minus sign (such as CR, DB, or the PL/I +).

## SIGNIFICANCE

**Purpose:** Control the significance (leading zero) indicator

**Format:** pattern

**Operation:**

```
PSW<C> ← 0;          !EO$CLEAR_SIGNIF
PSW<C> ← 1;          !EO$SET_SIGNIF
```

**Pattern Operators:**

| | | |
|---|---|---|
| 02 | EO$CLEAR_SIGNIF | Clear Significance |
| 03 | EO$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$SET_SIGNIF is used to avoid leading zero suppression (before EO$MOVE) or to force a fixed insert (before EO$INSERT).

## ADJUST INPUT LENGTH

**Purpose:** Handle source strings with lengths different from the output

**Format:** pattern len

**Operation:**

```
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<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 number in R0<31:16>.

**Note:** If length is not in the range 1 through 31 the destination string, condition codes, and R0 through R5 are unpredictable.

## END EDIT

**Purpose:** End the edit operation

**Format:** pattern

**Operation:**
```
if R0 NEQ 0 then {reserved operand}
if PSW<Z> EQL 1 then PSW<N> ← 0;
{end 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 12

# EXCEPTIONS

## 12.1 INTRODUCTION

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

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

Other events are primarily relevant to other processes, or to the system as a whole, and are therefore serviced in a system-wide context. The notification process for these events is termed an interrupt, and the system-wide context is described as "executing on the interrupt stack" (IS). Further, some interrupts are of such urgency that they require high-priority service, while others must be synchronized with independent events. To meet these needs, the processor has priority logic that grants interrupt service to the highest priority event at any point in time. The priority associated with an interrupt is termed its interrupt priority level (IPL). Interrupts are discussed in Volume 2.

Exceptions are handled by the operating system. Usually, they are reflected to the originating mode as a signal; see Appendix C. In general, the signal is described via a vector that is a counted list of longwords. The first longword contains the count of other longwords in the vector. The second longword identifies which exception occurred. The remaining longwords are the stack parameters, the PC, and the PSL, as described in this chapter.

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 the BISPSW and BICPSW instructions described in Chapter 7.

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 faulted 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

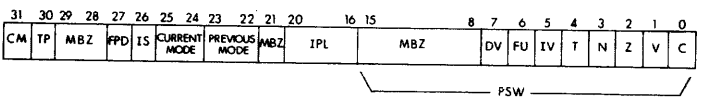
potentially leaves the registers and memory indeterminate, such that the instruction cannot necessarily be correctly restarted, completed, simulated, or undone.

## 12.2 PROCESSOR STATUS

When an exception or in interrupt is serviced, the processor status must be preserved so that the interrupted process may continue normally. Basically, this is done by automatically saving the Program Counter (PC) and the Processor Status Longword (PSL). These are later restored with the Return from Exception or Interrupt instruction (REI). Any other status required to correctly resume an interruptable instruction is stored in the general registers. Process context such as the mapping information is not saved or restored on each interrupt or exception. Instead, it is saved and restored only when process context switching is performed. Refer to the LDPCTX and SVPCTX instructions in Chapter 13. Other processor status is changed even less frequently; refer to the processor internal register descriptions in Chapter 13.

The Processor Status Longword (PSL) is a longword consisting of a word of privileged processor status concatenated with the Processor Status Word (PSW). Refer to Chapter 3 for a description of the PSW. The PSL is automatically saved on the stack when an exception or interrupt occurs and is saved in the Process Control Block on a process context switch. The PSL can also be stored by the MOVPSL instruction; refer to Chapter 7. (The terms current PSL and saved PSL are used to distinguish between this status information when it is in the processor and when copies of it are materialized in memory.)

Bits <31:21> of the current PSL can be changed explicitly only by executing a return from exception or interrupt instruction (REI). REI considers the current access mode when restoring the PSL, and faults if a program attempts to increase its privilege by this means. Thus REI is available to all software including user exception-handling routines.




Processor Status Longword

At bootstrap time, PSL is cleared except for IPL and IS.

| BITS | DESCRIPTION |
|---|---|
| 3:0 | Condition Codes: N, Z, V, C (See Chapter 3) |
| 4 | Trace enable (T). When set at the beginning of an instruction, causes TP to be set. When TP is set between instructions (before examining T), a trace fault is taken. The effect is that setting bit 4 forces a trace trap before the execution of each subsequent instruction. When clear, no trace exception occurs. Most programs should treat T as unpredictable because it is |

set by debuggers and trace programs for tracing and for proceeding from a breakpoint.

5 Integer Overflow trap enable (IV). When set, forces an integer overflow trap after execution of an instruction that produced an integer result that overflowed or had a conversion error. When IV is clear, no integer overflow trap occurs. (However, the condition code V bit is still set.)

6 Floating Underflow trap enable (FU). When set, forces a floating underflow trap after execution of an instruction that produced an underflowed result (i.e., a result exponent, after normalization and rounding, less than −127). When FU is clear, no trap occurs.

7 Decimal Overflow trap enable (DV). When set, forces a decimal overflow trap after execution of an instruction that produced an overflowed decimal (numeric string or packed decimal) result (i.e., no room to store a non-zero digit) or had a conversion error. When DV is clear, no trap occurs. (However, the condition code V bit is still set.)

15:8 Reserved to DIGITAL, must be zero.

20:16 Interrupt Priority Level (IPL). The current processor priority, in the range 0 to 31 (1F, hex). The processor will accept interrupts only on levels greater than the current level. At bootstrap time, IPL is initialized to 31 (1F, hex).

21 Reserved to DIGITAL, must be zero.

22:23 Previous Access Mode (PRVMOD). Loaded from current access mode by exceptions and CHMX instructions, cleared by interrupts, and restored by REI.

25:24 Current Access Mode (CURMOD). The access mode of the currently executing process, as follows:

0—KERNEL
1—EXECUTIVE
2—SUPERVISOR
3—USER

26 Interrupt Stack (IS). When set, the processor is executing on the interrupt stack. Any mechanism that sets IS also clears current access mode and raises IPL above 0. If an REI attempts to restore a PSL with IS = 1 and non-zero current access mode or zero IPL, a reserved operand fault is taken. When clear, the processor is executing on the stack specified by the current access mode. At bootstrap time, IS is set.

27 First Part Done (FPD). When set, the instruction addressed by PC cannot simply be restarted, and must be resumed at some other, implementation-specified, point in its operation. If FPD is set and the exception or interrupt service routine modifies FPD, the general registers, or the saved PSL (except for T or TP), the

results of the interrupted instruction's execution are unpredictable. If a routine sets FPD, the results are also unpredictable.

29:28 Reserved to DIGITAL, must be zero.

30 Trace Pending (TP). Forces a trace fault when set at the beginning of any instruction. Set by the processor if T is set at the beginning of an instruction. Any exception or interrupt service routine clearing TP must also clear T or the tracing of the interrupted instruction, if any, is unpredictable.

31 Compatibility Mode (CM). When set the processor is in PDP-11 compatibility mode, see Volume 2. When CM is clear, the processor is in native mode.

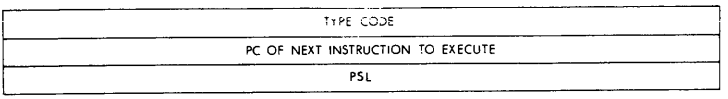
The software mnemonics for the PSL fields are:

| MNEMONIC | VALUE | MEANING |
|---|---|---|
| PSL$V_TBIT | 4 | position of trace enable |
| PSL$M_TBIT | 1 @ 4 | mask for trace enable |
| PSL$V_IV | 5 | position of IV enable |
| PSL$M_IV | 1 @ 5 | mask for IV enable |
| PSL$V_FU | 6 | position of FU enable |
| PSL$M_FU | 1 @ 6 | mask for FU enable |
| PSL$V_DV | 7 | position of DV enable |
| PSL$M_DV | 1 @ 7 | mask for DV enable |
| PSL$V_IPL | 16 | position of IPL |
| PSL$S_IPL | 5 | size of IPL |
| PSL$V_PRVMOD | 22 | position of PRVMOD |
| PSL$S_PRVMOD | 2 | size of PRVMOD |
| PSL$V_CURMOD | 24 | position of CURMOD |
| PSL$S_CURMOD | 2 | size of CURMOD |
| PSL$V_IS | 26 | position of IS bit |
| PSL$M_IS | 1 @ 26 | mask for IS bit |
| PSL$V_FPD | 27 | position of FPD bit |
| PSL$M_FPD | 1 @ 27 | mask for FPD bit |
| PSL$V_TP | 30 | position of TP bit |
| PSL$M_TP | 1 @ 30 | mask for TP bit |
| PSL$V_CM | 31 | position of CM bit |
| PSL$M_CM | 1 @ 31 | mask for CM bit |
| PSL$K_KERNEL | 0 | kernel mode |
| PSL$K_EXEC | 1 | executive mode |
| PSL$K_SUPER | 2 | supervisor mode |
| PSL$K_USER | 3 | user mode |

## 12.3 ARITHMETIC TRAPS

This section contains the descriptions of the exceptions that occur as the result of performing an arithmetic or conversion operation. They are mutually exclusive and all are assigned the same vector in the System Control Block, and hence the same signal "reason" code. Each of them indicates that an exception had occurred during the last instruc-

tion and that the instruction has been completed. The appropriate distinguishing code is pushed on the stack as a longword:




| TYPE CODE | TRAP TYPE | SOFTWARE MNEMONIC |
|---|---|---|
| 1 | integer overflow | SRM$K_INT_OVF_T |
| 2 | integer divide by zero | SRM$K_INT_DIV_T |
| 3 | floating overflow | SRM$K_FLT_OVF_T |
| 4 | floating/decimal divide by zero | SRM$K_FLT_DIV_T |
| 5 | floating underflow | SRM$K_FLT_UND_T |
| 6 | decimal overflow | SRM$K_DEC_OVF_T |
| 7 | subscript range | SRM$K_SUB_RNG_T |

### 12.3.1 Integer Overflow Trap

An integer overflow trap is an exception that indicates that the last instruction executed had an integer overflow setting the V condition code and that integer overflow was enabled (IV set). The result stored is the low-order part of the true result. N and Z are set according to the stored result. The type code pushed on the stack is 1 (SRM$K_INT_OVF_T). Note that the instructions RET, REI, REMQUE, MOVTUC, and BISPSW do not cause overflow even if they set V. Also note that the EMODx floating point instructions can cause integer overflow.

### 12.3.2 Integer Divide By Zero Trap

An integer divide by zero trap is an exception that indicates 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 (SRM$K_INT_DIV_T).

### 12.3.3 Floating Overflow Trap

A floating overflow trap is an exception that indicates that the last instruction executed resulted in an exponent greater than 127 (unbiased) after normalization and rounding. The result stored contains a one in the sign and zeros in the exponent and fraction fields. This is a reserved operand, and will cause a reserved operand fault if used in a subsequent floating point instruction. The N and V condition code bits are set and Z and C are cleared. The type code pushed on the stack is 3 (SRM$K_FLT_OVF_T).

### 12.3.4 Divide By Zero Trap — Floating or Decimal String

A floating divide by zero trap is an exception that indicates that the last instruction executed had a floating zero divisor. The result stored is the reserved operand, as described above for floating overflow trap, and the condition codes are set as in floating overflow.

A decimal string divide by zero trap is an exception that indicates 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 for both types of divide by zero is 4 (SRM$K_FLT_DIV_T).

### 12.3.5 Floating Underflow Trap

A floating underflow trap is an exception that indicates that the last instruction executed resulted in an exponent less than −127 (unbiased) after normalization and rounding and that floating underflow was enabled (FU set). The result stored is zero. Except for POLYx, the N, V, and C condition codes are cleared and Z is set. In POLYx, the trap occurs on completion of the instruction, which may be many operations after the underflow. The condition codes are set on the final result in POLYx. The type code pushed on the stack is 5 (SRM$K_FLT_UND_T).

### 12.3.6 Decimal String Overflow Trap

A decimal string overflow trap is an exception that indicates that the last instruction executed had a decimal string result too large for the destination string provided and that decimal overflow was enabled (DV set). The V condition code is always set. Refer to the individual instruction descriptions for the value of the result and of the condition codes. The type code pushed on the stack is 6 (SRM$K_DEC_OVF_T).

### 12.3.7 Subscript Range Trap

A subscript range trap is an exception that indicates 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 (SRM$K_SUB_RNG_T).

## 12.4 EXCEPTIONS DETECTED DURING OPERAND

### 12.4.1 Access Control Violation Fault

An access control violation fault is an exception indicating that the process attempted a reference not allowed at the access mode at which the process was operating. See Volume 2 for a description of the information pushed on the stack as parameters. Software may restart the process after changing the address translation information.

### 12.4.2 Translation Not Valid Fault

A translation not valid fault is an exception indicating that the process attempted a reference to a page for which the valid bit in the page table was not set. See Volume 2 for a description of the information pushed on the stack as parameters. Note that if a process attempts to reference a page for which the page table entry specifies both not valid and access violation, an access control violation fault occurs.

### 12.4.3 Reserved Addressing Mode Fault

A reserved addressing mode fault is an exception indicating that an operand specifier attempted to use an addressing mode that is not allowed in the situation in which it occurred. No parameters are pushed.

The situations in which each specifier type is reserved are:

| SPECIFIER | RESERVED SITUATION |
|---|---|
| Short Literal | Modify, destination, address |
| Register | source, or within index mode. |
| Index Mode | Address source or within index mode. Within index mode, or with PC as index. |

See Chapter 5 for combinations of addressing modes and registers that cause unpredictable results. The VAX-11/780 processor also faults on PC, @PC, and —(PC).

### 12.4.4 Reserved Operand Exception

A reserved operand exception is an exception indicating that an operand accessed has a format reserved for future use by DIGITAL. No parameters are pushed. This exception always backs up the PC to point to the opcode. The exception service routine may determine the type of operand by examining the opcode using the stored PC. Note that only the changes made by instruction fetch and because of operand specifier evaluation may be restored. Therefore, some instructions are not restartable. These exceptions are labelled as ABORTs rather than FAULTs. The PC is always restored properly unless the instruction attempted to modify it in a manner that results in unpredictable results. The PSL other than FPD and TP is not changed except for the condition codes, which are unpredictable.

The reserved operand exceptions are caused by:

1. A floating point number that has the sign bit set and the exponent zero except in the POLY table (FAULT)
2. A floating point number that has the sign bit set and the exponent zero in the POLY table (ABORT, see chapter 6 for restartability)
3. POLY degree too large (FAULT)
4. Bit field too wide (FAULT)
5. Invalid CALLx entry mask (FAULT)
6. Invalid combination of bits in PSW/MASK longword during RET (FAULT)
7. Invalid combination of bits in BISPSW/BICPSW (FAULT)
8. Unaligned operand in ADAWI (FAULT)
9. Unaligned queue entry or header in INSQUE or REMQUE (FAULT)
10. Decimal string too long (FAULT)
11. Invalid digit in CVTTP, CVTSP (FAULT)
12. Reserved pattern operator in EDITPC (ABORT, see Chapter 11 for restartability)
13. Incorrect source string length at completion of EDITPC (ABORT)
14. Invalid combination of bits in PSL restored by REI (FAULT)
15. Invalid register number in MFPR or MTPR (FAULT)
16. Invalid register contents in some MTPR (FAULT)
17. Invalid combinations in Process Control Block loaded by LDPCTX (ABORT)

## 12.5 EXCEPTIONS OCCURRING AS THE CONSEQUENCE OF AN INSTRUCTION

### 12.5.1 Opcode Reserved To DIGITAL Fault

An opcode reserved to DIGITAL fault occurs when the processor encounters an opcode that is not specifically defined, or that requires

higher privileges than the current access mode. No parameters are pushed. Opcode FFFF (hex) will always fault.

### 12.5.2 Opcode Reserved To Customers (and CSS) Fault

An opcode reserved to customers fault is an exception that occurs when an opcode reserved to the customers or DIGITAL's Computer Special Systems group is executed. The operation is identical to the opcode reserved to DIGITAL fault except that the event is caused by a different set of opcodes, and faults through a different vector. All opcodes reserved to customers (and CSS) start with FC (hex) which is the XFC instruction. If the special instruction needs to generate a unique exception, one of the reserved to CSS/customer vectors in the System Control Block should be used. An example might be that the particular second byte of the instruction opcode is not recognized or implemented by the hardware.

### 12.5.3 Compatibility Mode Exception

A compatibility mode exception is an exception that occurs when the processor is in compatibility mode. See Volume 2 for details.

### 12.5.4 Breakpoint Fault

A breakpoint fault is an exception that occurs when the breakpoint instruction (BPT) is executed. No parameters are pushed.

To proceed from a breakpoint, a debugger or tracing program typically restores the original contents of the location containing the BPT, sets T in the PSL saved by the BPT fault, and resumes. When the breakpoint instruction completes, a trace execption will occur; see section 12.6. At this point, the tracing program can again re-insert the BPT instruction, restore T to its original state (usually clear), and resume. Note that if both tracing and breakpointing are in progress (i.e., if PSL<T> was set at the time of the BPT), then on the trace exception both the BPT restoration and a normal trace exception should be processed by the trace handler.

## 12.6 TRACING

A trace is an exception that occurs between instructions when trace is enabled. Tracing is used for tracing programs, for performance evaluation, or debugging purposes. It is designed so that one and only one trace exception occurs before the execution of each traced instruction (except that a service routine invoked by CHMx and terminated by REI is considered a single instruction). The saved PC on a trace is the address of the next instruction that would normally be executed.

In order to ensure that exactly one trace occurs per instruction despite other traps and faults, the PSL contains two bits, trace enable (T) and trace pending (TP); see section 12.2. If only one bit were used then the occurrence of an interrupt at end of instruction would either produce zero or two traces, depending on the design. Instead, the PSL<T> bit is defined to produce a trap after any other traps or aborts. The trap effect is implemented by copying PSL<T> to a second bit (PSL<TP>) that is actually used to generate the exception. PSL<TP> generates a fault before any other processing at the start of the next instruction. See Volume 2 for detailed flows.

The rules of operation for trace are:

1. At the beginning of an instruction, if T is set, then TP is set.
2. If the instruction faults or an interrupt is serviced, the pushed PSL<TP> is cleared. The pushed PC is set to the start of the faulting or interrupted instruction.
3. If the instruction aborts or takes an arithmetic trap, the pushed PSL<TP> is set or cleared as the result of step 1.
4. If an interrupt is serviced after instruction completion and arithmetic traps, but before tracing is checked for at the start of the next instruction, then the pushed PSL<TP> is set or cleared as the result of step 1.
5. At the beginning of an instruction, if TP is set, then a trace pending fault is taken.

The routine entered by a CHMx is not traced because CHMx clears T and TP in the new PSL. However, if T was set at the beginning of CHMx, the saved PSL will have both T and TP set. REI will trap either if T was set when the REI was executed or if TP in the saved PSL is set. Because of this, the instruction sequence CHMx . . . REI acts as a single instruction. Note that the trace exception occurring after an REI that had TP set before being executed will be taken with the new PSL. Thus, special care must be taken if exception or interrupt routines are traced.

In addition, the CALLx instructions save a clear T, although T in the PSL is unchanged. This is done so that a debugger or trace program proceeding from a BPT fault does not get a spurious trace from the RET that matches the CALL; see 12.5.4.

The detection of interrupts and other exceptions occurs before the detection of a trace exception. However, this causes no difficulties since the entire PSL (including T and TP) is automatically saved on interrupt or exception initiation and is restored at the end with an REI. This makes interrupts and benign exceptions totally transparent to the executing program.

### 12.6.1 Trace Instruction Summary

The following table shows all of the cases of T enabled at the beginning of the instruction, enabled at the end of the instruction, and TP set in the popped PSW or PSL for ordinary instructions (XXX), CHMx . . . REI, interrupt or exception . . . REI, CALLx, RETURN, CHMx, REI, BISPSW, and BICPSW:

| | TRACE EXCEPTION | | |
|---|---|---|---|
| | enabled at beg (T) | enabled at end (T) | TP bit at end (TP) |
| XXX | N | N | N |
| | Y | Y | Y |
| CHMx . . . REI | N | N | N |
| | Y | Y | Y |

| | | | | |
|---|---|---|---|---|
| interrupt or exception . . . REI | N<br>Y | N<br>Y | N<br>Y | |
| CALLx | N<br>Y | N<br>Y | N<br>Y | (pushed PSW<T> clear) |
| RET | N<br>N<br>Y<br>Y | N*<br>Y*<br>N*<br>Y* | N<br>N<br>Y<br>Y | (trap after next instruction) |
| CHMx | N<br>Y | N<br>N | N<br>N | (pushed PSL<TP> clear)<br>(pushed PSL<TP> set) |
| REI (if PSL<TP>=0 on stack) | N<br>N<br>Y<br>Y | N*<br>Y*<br>N*<br>Y* | N<br>N<br>Y<br>Y | |
| REI (if PSL<TP>=1 on stack) | N<br>N<br>Y<br>Y | N*<br>Y*<br>N*<br>Y* | Y<br>Y<br>Y<br>Y | (only one trap) |
| BISPSW | N<br>Y | Y<br>Y | N<br>Y | |
| BICPSW | N<br>Y | N<br>N | N<br>Y | |
| interrupt or exception | N<br>Y | N<br>N | N<br>N | (pushed PSL<TP> clear)<br>(pushed PSL<TP> depends on above description) |

* = depends on PSW<T> popped from stack

### 12.6.2 Using Trace

Routines using the trace facility are termed trace handlers. They should observe the following conventions and restrictions:

1. When the trace handler performs its REI back to the traced program, it should always force the T bit on in the PSL that will be restored. This defends against programs clearing T via RET, REI, or BICPSW.
2. The trace handler should never examine or alter the TP bit when continuing tracing. The hardware flows ensure that this bit is maintained correctly to continue tracing.
3. When tracing is to be ended, both T and TP should be cleared. This ensures that no further traces will occur.
4. Tracing a service routine that completes with an REI will give a trace in the restored mode after the REI. If the program being restored to was also being traced, only one trace exception is generated.

5. If a routine entered by a CALLx instruction is executed at full speed by turning off T, then trace control can be regained by setting T in the PSW in its call frame. Tracing will resume after the instruction following the RET.
6. Tracing is disabled for routines entered by a CHMx instruction or any exception. Thus, if a CHMx or exception service routine is to be traced, a breakpoint instruction must be placed at its entry point. If such a routine is recursive, breakpointing will catch each recursion only if the breakpoint is not on the CHMx or the instruction with the exception.
7. If it is desired to allow multiple trace handlers, all handlers should preserve T when turning on and off trace. They also would have to simulate traced code that alters or reads T.

## 12.7 SERIOUS SYSTEM FAILURES

Refer to Volume 2 for details on the following failures.

### 12.7.1 Kernel Stack Not Valid Abort

Kernel stack not valid abort is an exception that indicates that the kernel stack was not valid while the processor was pushing information onto the kernel stack during the initiation of an exception or interrupt.

### 12.7.2 Interrupt Stack Not Valid Halt

An interrupt stack not valid halt is an exception that indicates that the interrupt stack was not valid or that a memory error occurred while the processor was pushing information onto the interrupt stack during the initiation of an exception or interrupt.

### 12.7.3 Machine Check Exception

A machine check exception indicates that the processor detected an internal error in itself.

## 12.8 STACKS

At any time, the processor is either in a process context (IS = 0) in one of four modes (kernel, exec, super, user), or in the system-wide interrupt service context (IS = i) that operates with kernel privileges. There is a stack pointer associated with each of these five states, and any time the processor changes from one of these states to another, SP (R14) is stored in the process context stack pointer for the old state and loaded from that for the new state. The process context stack pointers (KSP = kernel, ESP = exec, SSP = super, USP = user) are allocated in the Process Control Block; see Volume 2; although some hardware implementations may keep them in internal registers. The interrupt stack pointer (ISP) is in an internal register.

### 12.8.1 Stack Residency

The user, super, and exec stacks do not need to be resident. The kernel can bring in or allocate process stack pages as address translation not valid faults occur. However, the kernel stack for the current process, and the interrupt stack (which is process-independent) must be resident and accessible. Translation not valid and access control violation faults occurring on references to either of these stacks are regarded as serious system failures, from which recovery is not possible.

If either of these faults occurs on a reference to the kernel stack, the processor aborts the current sequence and initiates kernel stack not valid abort on hardware level 31 (1F, hex). If either of these faults occurs on a reference to the interrupt stack, the processor halts. Note that this does not mean that every possible reference is checked, but rather that the processor will not loop on these conditions.

It is not necessary that the kernel stack for processes other than the current one be resident, but it must be resident before a process is selected to run by the software's process dispatcher. Further, any mechanism that uses translation not valid or access control violation faults to gather process statistics, for instance, must exercise care not to invalidate kernel stack pages.

### 12.8.2 Stack Alignment

Except on CALLx instructions, the hardware makes no attempt to align the stacks. For best performance on all processors, the software should align the stack on a longword boundary and allocate the stack in longword increments. The convert byte to long (CVTBL and MOVZBL), convert word to long (CVTWL and MOVZWL), convert long to byte (CVTLB), and convert long to word (CVTLW) instructions are recommended for pushing bytes and words on the stack and popping them off in order to keep the stack longword aligned.

### 12.8.3 Stack Status Bits

The interrupt stack bit (IS) and current access mode bits in the privileged Processor Status Longword (PSL) specify which of the five stack pointers is currently in use as follows:

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

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

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

# REI RETURN FROM EXCEPTION OR INTERRUPT

## 12.9 RELATED INSTRUCTIONS

**Purpose:** exit from an execption or interrupt service routine

**Format:** Opcode

**Operation:**

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

if (tmp2<current_mode> LSSU<current_mode>) or
   (tmp2<IS> EQLU 1 and PSL<IS> EQLU 0) or
   (tmp2<IS> EQLU 1 and
   tmp2<current_mode> NEQU 0) or
   (tmp2<IS> EQLU 1 and tmp2<IPI> EQLU 0) or
   (tmp2<IPL> GTRU 0 and
   tmp2<current_mode> NEQU 0) or
   (tmp2<previous_mode> LSSU
   temp2<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);

(disallow interrupts);
if PSL<IS> EQLU 1 then ISP ← SP
                                    !save old stack pointer
     else PSL<current_mode>_SP ← SP;
if PSL<TP> EQLU 1 then tmp2<TP> ← 1;
                                     !TP ← TP or stack TP
PC ← tmp1;
PSL ← tmp2;
if PSL<IS> EQLU 0 then
     begin
     SP ← PSL<current_mode>_SP;        !switch stack
     if PSL<current_mode> GEQU ASTLVL
                                    !check for AST delivery
     then (request interrupt at IPL 2);
end;
(allow interrupts);
```

**Condition Codes:**

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

**Exceptions:** reserved operand

**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, see 12.8.3. The level of the highest privilege AST is checked against the current access mode to see whether a pending AST can be delivered; refer to Volume 2. 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 non-interrupt 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; see Chapter 13. MFPR and MTPR always fetch and store the pointers whether in registers or the Process Control Block.

## BPT BREAKPOINT FAULT

**Purpose:** stop for debugging

**Format:** Opcode

**Operation:**
PSL<TP> ← 0;
{breakpoint fault};

**Condition Codes:**
N ← 0;
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.

## HALT

**Purpose:** stop processor operation

**Format:** Opcode

**Operation:**

```
If PSL<current_mode> NEQU kernel then
    {reserved to DIGITAL opcode fault}
    else
    {halt the processor};
```

**Condition Codes:**

```
N ← 0;    !If reserved to DIGITAL opcode fault
Z ← 0;
V ← 0;
C ← 0;

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

**Exceptions:** reserved to DIGITAL opcode

**Opcodes:** 00 HALT Halt

**Description:** If the process is running in kernel mode, the processor is halted. Otherwise, a reserved to DIGITAL opcode fault occurs.

**Note:** This opcode is 0 to trap many branches to data.

# CHAPTER 13

# PRIVILEGE INSTRUCTIONS

The privilege instructions allow access to privileged operations within the VAX-11 system. The change mode instructions provide a controlled mechanism for unprivileged software to request services of more privileged software. In particular, the change mode instructions are the only normal way for code executing at executive, supervisor, or user access modes to change to a more privileged mode. In all cases, the change mode results in transferring control to a fixed location depending upon contents of the System Control Block. See Chapter 12 for a discussion of change mode exception handling.

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

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

The move to and from processor register instructions provides software executing in kernel mode access to the internal control registers of the processor. This allows such operations as control of the memory management system and selection of the address of the Process Control Block of the next process to execute. The load and save process context instructions allow kernel mode software to save and restore the general register and memory management status of a process when switching between processes. The processor register and process context instructions are described more fully in volume 2 of the Processor Handbook.

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

## CHANGE MODE

**Purpose:** request services of more privileged software

**Format:** opcode code.rw

**Operation:**

```
if (PSL<IS> EQLU 1) then HALT;          !illegal from
                                        Interrupt stack
(switch stack pointer from current-mode to MINU (op-
  code-mode, PSL<current-mode>) );
-(SP)    ← PSL;                         !initiate CHMx
-(SP)    ← PC;                          exception
-(SP)    ← SEXT (code);
PSL<CM, TP, FPD, DV, FU, IV, T, N, Z, V, C>  ← 0;
  !clean out PSL
PSL<previous-mode>   ← PSL<current-mode>;
PSL<current-mode>   ← MINU (opcode-mode, PSL<cur-
  rent-mode>);                          !maximize
                                        privilege
PC <— (SCB vector for opcode-mode);
```

**Condition Codes:**

```
Z ← 0;
N ← 0;
V ← 0;
C ← 0;
```

**Exceptions:** halt

**Opcodes:**

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

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

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

| | |
|---|---|
| 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.

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

**Examples:**

```
CHMK    #7      ;request the kernel mode service
                ;specified by code 7
CHME    #4      ;request the executive mode service
                ;specified by code 4
CHMS    #—2     ;request the supervisor mode service
                ;specified by customer code —2
```

## PROBE ACCESSIBILITY

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

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

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

**Condition Codes:**
N ← 0;
Z ← if both accessible then 0; else 1;
V ← 0;
C ← 0;

**Exceptions:** translation not valid

**Opcodes:**

| | | |
|---|---|---|
| 0C | PROBER | Probe Read Accessibility |
| 0D | PROBEW | Probe Write Accessibility |

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

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

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

**Examples:**

```
MOVL    4(AP),R0        ;copy address of first arg so
                        ;that it can't be changed
PROBER  #0,#4,R0        ;verify that the longword pointed
                        ;to by the first argument could be
                        ;read by the previous access
                            mode
                        ;Note that the argument list
                            itself
                        ;must already have been probed
```

```
MOVQ    8(AP),R0        ;copy length and address
                        ;of buffer arguments so that
                        ;they can't change
PROBEW  #0,R0,R1        ;verify that the buffer described
                        ;by the second and third argu-
                            ments
                        ;could be written by the previous
                        ;access mode
                        ;Note that the argument list must
                        ;already have been probed and
                            that
                        ;the second argument must be
                            known
                        ;to be less than 514
```

## EXTENDED FUNCTION CALL

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

**Format:** opcode

**Operation:** {XFC fault};

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

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

**Opcodes:** FC XFC Extended Function Call

**Description:** This instruction requests services of non-standard microcode or software. If no special microcode is loaded then an exception is generated to a kernel mode software simulator (see Chapter 12). Typically, the next byte would specify which of several extended functions are requested. Parameters would be passed either as normal operands, or more likely in fixed registers.

## MOVE TO PROCESSOR REGISTER
## MOVE FROM PROCESSOR REGISTER

**Purpose:** provide access to the internal processor registers

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

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

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

**Exceptions:**
reserved operand
reserved instruction

**Opcode:**

| | | |
|---|---|---|
| DA | MTPR | Move To Processor Register |
| DB | MFPR | Move From Processor Register |

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

**Notes:**

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

The following table is a summary of the registers accessible in the processor register space. For information on the processor registers, refer to Volume 2.

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 pre-defined 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 processor register present on more than one implementation will perform the same function whenever implemented. All unassigned positive numbers are reserved to DIGITAL; all negative numbers (i.e., with bit 31 set) are reserved to CSS and customers.

**VAX-11 Series Registers**

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

**VAX-11/780 Specific Registers**

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

## LOAD PROCESS CONTEXT
## SAVE PROCESS CONTEXT

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

**Format:** opcode

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

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

**Exceptions:** reserved operand
reserved instruction

**Opcodes:**

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

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

## APPENDIX A

# DATA TABLES

### A.1 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

*American Standard Code for Information Interchange.

### A.2 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 |

### A.3 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 (0). Add the hexadecimal equivalents to obtain the hexadecimal value.

Example:

| | | | |
|---|---|---|---|
| 22,466(10) | = | ?(16) | |
| 20,480 | = | 5000 | 22,466 |
| 1,792 | = | 700 | -20,480 |
| 192 | = | C0 | |
| 2 | = | 2 | 1,986 |
| | = | | -1,792 |
| 22,466 | = | 57C2 | |
| | | | 194 |
| | | | -192 |
| | | | 2 |
| | | | -2 |
| | | | 0 |

## A.4 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 hex.

## A.5 HEXADECIMAL MULTIPLICATION

Table A-3 is 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, loctae 4 in the left-hand column and A in the topmost row. The intersection of the two is Z8 hex which is the product of the two numbers.

**Table A-1 HEXADECIMAL INTEGER COLUMNS**

| | 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 LONGWORD 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 | )D | 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 | 69 |
| 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 |

## A.6 ASCII CHARACTER SET AND HEX-ASCII CONVERSION

Table A-4 is a table representing 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 | ESC | 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 |

## A.7 POWERS OF 2 AND POWERS OF 16

For quick reference, the most commonly used powers of 2 and powers 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

## B.1. MNEMONIC LISTING

| MNEMONIC | INSTRUCTION | OPCODE | PAGE |
|---|---|---|---|
| ACBB | Add compare and branch byte | 9D | 8-10 |
| ACBD | Add compare and branch double | 6F | 8-10 |
| ACBF | Add compare and branch floating | 4F | 8-10 |
| ACBL | Add compare and branch long | F1 | 8-10 |
| ACBW | Add compare and branch word | 3D | 8-10 |
| ADAWI | Add aligned word interlocked | 58 | 6-21 |
| ADDB2 | Add byte 2 operand | 80 | 6-18 |
| ADDB3 | Add byte 3 operand | 81 | 6-18 |
| ADDD2 | Add double 2 operand | 60 | 6-18 |
| ADDD3 | Add double 3 operand | 61 | 6-18 |
| ADDF2 | Add floating 2 operand | 40 | 6-18 |
| ADDF3 | Add floating 3 operand | 41 | 6-18 |
| ADDL2 | Add long 2 operand | C0 | 6-18 |
| ADDL3 | Add long 3 operand | C1 | 6-18 |
| ADDP4 | Add packed 4 operand | 20 | 10-6 |
| ADDP6 | Add packed 6 operand | 21 | 10-6 |
| ADDW2 | Add word 2 operand | A0 | 6-18 |
| ADDW3 | Add word 3 operand | A1 | 6-18 |
| ADWC | Add with carry | D8 | 6-20 |
| AOBLEQ | Add one and branch on less or equal | F3 | 8-12 |
| AOBLSS | Add one and branch on less | F2 | 8-12 |
| ASHL | Arithmetic shift long | 78 | 6-37 |
| ASHP | Arithmetic shift and round packed | F8 | 10-22 |
| ASHQ | Arithmetic shift quad | 79 | 6-37 |
| BBC | Branch on bit clear | E1 | 8-5 |
| BBCC | Branch on bit clear and clear | E5 | 8-6 |
| BBCCI | Branch on bit clear and clear interlocked | E7 | 8-7 |
| BBCS | Branch on bit clear and set | E3 | 8-6 |
| BBS | Branch on bit set | E0 | 8-5 |
| BBSC | Branch on bit set and clear | E4 | 8-6 |
| BBSS | Branch on bit set and set | E2 | 8-6 |
| BBSSI | Branch on bit set and set interlocked | E6 | 8-7 |
| BCC | Branch on carry clear | 1E | 8-2 |
| BCS | Branch on carry set | 1F | 8-2 |
| BEQL | Branch on equal | 13 | 8-2 |
| BEQLU | Branch on equal unsigned | 13 | 8-2 |
| BGEQ | Branch on greater or equal | 18 | 8-2 |
| BGEQU | Branch on greater or equal unsigned | 1E | 8-2 |

| MNEMONIC | INSTRUCTION | OPCODE | PAGE |
|---|---|---|---|
| BGTR | Branch on greater | 14 | 8-2 |
| BGTRU | Branch on greater unsigned | 1A | 8-2 |
| BICB2 | Bit clear byte 2 operand | 8A | 6-35 |
| BICB3 | Bit clear byte 3 operand | 8B | 6-35 |
| BICL2 | Bit clear long 2 operand | CA | 6-35 |
| BICL3 | Bit clear long 3 operand | CB | 6-35 |
| BICPSW | Bit clear program status word | B9 | 7-5 |
| BICW2 | Bit clear word 2 operand | AA | 6-35 |
| BICW3 | Bit clear word 3 operand | AB | 6-35 |
| BISB2 | Bit set byte 2 operand | 88 | 6-34 |
| BISB3 | Bit set byte 3 operand | 89 | 6-34 |
| BISL2 | Bit set long 2 operand | C8 | 6-34 |
| BISL3 | Bit set long 3 operand | C9 | 6-34 |
| BISPSW | Bit set program status word | B8 | 7-5 |
| BISW2 | Bit set word 2 operand | A8 | 6-34 |
| BISW3 | Bit set word 3 operand | A9 | 6-34 |
| BITB | Bit test byte | 93 | 6-33 |
| BITL | Bit test long | D3 | 6-33 |
| BITW | Bit test word | B3 | 6-33 |
| BLBC | Branch on low bit clear | E9 | 8-8 |
| BLBS | Branch on low bit set | E8 | 8-8 |
| BLEQ | Branch on less or equal | 15 | 8-2 |
| BLEQU | Branch on less or equal unsigned | 1B | 8-2 |
| BLSS | Branch on less | 19 | 8-2 |
| BLSSU | Branch on less unsigned | 1F | 8-2 |
| BNEQ | Branch on not equal | 12 | 8-2 |
| BNEQU | Branch on not equal unsigned | 12 | 8-2 |
| BPT | Break point fault | 03 | 12-15 |
| BRB | Branch with byte displacement | 11 | 8-4 |
| BRW | Branch with word displacement | 31 | 8-4 |
| BSBB | Branch to subroutine with byte displacement | 10 | 8-16 |
| BSBW | Branch to subroutine with word displacement | 30 | 8-16 |
| BVC | Branch on overflow clear | 1C | 8-2 |
| BVS | Branch on overflow set | 1D | 8-2 |
| CALLG | Call with general argument list | FA | 8-20 |
| CALLS | Call with stack | FB | 8-22 |
| CASEB | Case byte | 8F | 8-14 |
| CASEL | Case long | CF | 8-14 |
| CASEW | Case word | AF | 8-14 |
| CHME | Change mode to executive | BD | 13-2 |
| CHMK | Change mode to kernel | BC | 13-2 |
| CHMS | Change mode to supervisor | BE | 13-2 |
| CHMU | Change mode to user | BF | 13-2 |
| CLRB | Clear byte | 94 | 6-9 |
| CLRD | Clear double | 7C | 6-9 |

| MNEMONIC | INSTRUCTION | OPCODE | PAGE |
|---|---|---|---|
| CLRF | Clear float | D4 | 6-9 |
| CLRL | Clear long | D4 | 6-9 |
| CLRQ | Clear quad | 7C | 6-9 |
| CLRW | Clear word | B4 | 6-9 |
| CMPB | Compare byte | 91 | 6-15 |
| CMPC3 | Compare character 3 operand | 29 | 9-8 |
| CMPC5 | Compare character 5 operand | 2D | 9-8 |
| CMPD | Compare double | 71 | 6-15 |
| CMPF | Compare floating | 51 | 6-15 |
| CMPL | Compare long | D1 | 6-15 |
| CMPP3 | Compare packed 3 operand | 35 | 10-5 |
| CMPP4 | Compare packed 4 operand | 37 | 10-5 |
| CMPV | Compare field | EC | 7-20 |
| CMPW | Compare word | B1 | 6-15 |
| CMPZV | Compare zero-extended field | ED | 7-20 |
| CRC | Calculate cyclic redundancy check | 0B | 9-14 |
| CVTBD | Convert byte to double | 6C | 6-12 |
| CVTBF | Convert byte to float | 4C | 6-12 |
| CVTBL | Convert byte to long | 98 | 6-12 |
| CVTBW | Convert byte to word | 99 | 6-12 |
| CVTDB | Convert double to byte | 68 | 6-12 |
| CVTDF | Convert double to float | 76 | 6-12 |
| CVTDL | Convert double to long | 6A | 6-12 |
| CVTDW | Convert double to word | 69 | 6-12 |
| CVTFB | Convert float to byte | 48 | 6-12 |
| CVTFD | Convert float to double | 56 | 6-12 |
| CVTFL | Convert float to long | 4A | 6-12 |
| CVTFW | Convert float to word | 49 | 6-12 |
| CVTLB | Convert long to byte | F6 | 6-12 |
| CVTLD | Convert long to double | 6E | 6-12 |
| CVTLF | Convert long to float | 4E | 6-12 |
| CVTLP | Convert long to packed | F9 | 6-12 |
| CVTLW | Convert long to word | F7 | 6-12 |
| CVTPL | Convert packed to long | 36 | 10-14 |
| CVTTP | Convert trailing numeric to packed | 26 | 10-17 |
| CVTPT | Convert packed to trailing numeric | 24 | 10-15 |
| CVTPS | Convert packed to leading separate numeric | 08 | |
| CVTRDL | Convert rounded double to long | 6B | 6-12 |
| CVTRFL | Convert rounded float to long | 4B | 6-12 |
| CVTSP | Convert leading separate numeric to packed | 09 | |
| CVTWB | Convert word to byte | 33 | 6-12 |
| CVTWD | Convert word to double | 6D | 6-12 |
| CVTWF | Convert word to float | 4D | 6-12 |
| CVTWL | Convert word to long | 32 | 6-12 |
| DECB | Decrement byte | 97 | 6-24 |
| DECL | Decrement long | D7 | 6-24 |
| DECW | Decrement word | B7 | 6-24 |

| MNEMONIC | INSTRUCTION | OPCODE | PAGE |
|---|---|---|---|
| DIVB2 | Divide byte 2 operand | 86 | 6-30 |
| DIVB3 | Divide byte 3 operand | 87 | 6-30 |
| DIVD2 | Divide double 2 operand | 66 | 6-30 |
| DIVD3 | Divide double 3 operand | 67 | 6-30 |
| DIVF2 | Divide floating 2 operand | 46 | 6-30 |
| DIVF3 | Divide floating 3 operand | 47 | 6-30 |
| DIVL2 | Divide long 2 operand | C6 | 6-30 |
| DIVL3 | Divide long 3 operand | C7 | 6-30 |
| DIVP | Divide packed | 27 | 10-11 |
| DIVW2 | Divide word 2 operand | A6 | 6-30 |
| DIVW3 | Divide word 3 operand | A7 | 6-30 |
| EDITPC | Edit packed to character | 38 | |
| EDIV | Extended divide | 7B | 6-32 |
| EMODD | Extended modulus double | 74 | 6-29 |
| EMODF | Extended modulus floating | 54 | 6-29 |
| EMUL | Extended multiply | 7A | 6-28 |
| EXTV | Extract field | EE | 7-18 |
| EXTZV | Extract zero-extended field | EF | 7-18 |
| FFC | Find first clear bit | EB | 7-16 |
| FFS | Find first set bit | EA | 7-16 |
| HALT | Halt | 00 | 12-16 |
| INCB | Increment byte | 96 | 6-16 |
| INCL | Increment long | D6 | 6-16 |
| INCW | Increment word | B6 | 6-16 |
| INDEX | Compute index | 0A | 7-8 |
| INSQUE | Insert into queue | 0E | 7-12 |
| INSV | Insert field | F0 | 7-22 |
| JMP | Jump | 17 | 8-4 |
| JSB | Jump to subroutine | 16 | 8-16 |
| LDPCTX | Load process context | 16 | 13-10 |
| LOCC | Locate character | 3A | 9-11 |
| MATCHC | Match characters | 39 | 9-12 |
| MCOMB | Move complemented byte | 92 | 6-11 |
| MCOML | Move complemented long | D2 | 6-11 |
| MCOMW | Move complemented word | B2 | 6-11 |
| MFPR | Move from processor register | DB | 13-7 |
| MNEGB | Move negated byte | 8E | 6-10 |
| MNEGD | Move negated double | 72 | 6-10 |
| MNEGF | Move negated floating | 52 | 6-10 |
| MNEGL | Move negated long | CE | 6-10 |
| MNEGW | Move negated word | AE | 6-10 |
| MOVAB | Move address of byte | 9E | 7-6 |
| MOVAD | Move address of double | 7E | 7-6 |
| MOVAF | Move address of float | DE | 7-6 |
| MOVAL | Move address of long | DE | 7-6 |

| MNEMONIC | INSTRUCTION | OPCODE | PAGE |
|---|---|---|---|
| MOVAQ | Move address of quad | 7E | 7-6 |
| MOVAW | Move address of word | 3E | 7-6 |
| MOVB | Move byte | 90 | 6-7 |
| MOVC3 | Move character 3 operand | 28 | 9-2 |
| MOVC5 | Move character 5 operand | 2C | 9-2 |
| MOVD | Move double | 70 | 6-7 |
| MOVF | Move float | 50 | 6-7 |
| MOVL | Move long | D0 | 6-7 |
| MOVP | Move packed | 34 | 10-4 |
| MOVPSL | Move processor status longword | DC | 7-4 |
| MOVQ | Move quad | 7D | 6-7 |
| MOVTC | Move translated characters | 2E | 9-4 |
| MOVTUC | Move translated until character | 2F | 9-6 |
| MOVW | Move word | B0 | 6-7 |
| MOVZBL | Move zero-extended byte to long | 9A | 6-14 |
| MOVZBW | Move zero-extended byte to word | 9B | 6-14 |
| MOVZWL | Move zero-extended word to long | 3C | 6-14 |
| MTPR | Move to processor register | DA | 13-7 |
| MULB2 | Multiply byte 2 operand | 84 | 6-26 |
| MULB3 | Multiply byte 3 operand | 85 | 6-26 |
| MULD2 | Multiply double 2 operand | 64 | 6-26 |
| MULD3 | Multiply double 3 operand | 65 | 6-26 |
| MULF2 | Multiply floating 2 operand | 44 | 6-26 |
| MULF3 | Multiply floating 3 operand | 45 | 6-26 |
| MULL2 | Multiply long 2 operand | C4 | 6-26 |
| MULL3 | Multiply long 3 operand | C5 | 6-26 |
| MULP | Multiply packed | 25 | 10-10 |
| MULW2 | Multiply word 2 operand | A4 | 6-26 |
| MULW3 | Multiply word 3 operand | A5 | 6-26 |
| NOP | No operation | 01 | |
| POLYD | Evaluate polynomial double | 75 | 6-39 |
| POLYF | Evaluate polynomial floating | 55 | 6-39 |
| POPR | Pop registers | BA | |
| PROBER | Probe read access | 0C | 13-4 |
| PROBEW | Probe write access | 0D | 13-4 |
| PUSHAB | Push address of byte | 9F | 7-6 |
| PUSHAD | Push address of double | 7F | 7-6 |
| PUSHAF | Push address of float | DF | 7-6 |
| PUSHAL | Push address of long | DF | 7-6 |
| PUSHAQ | Push address of quad | 7F | 7-6 |
| PUSHAW | Push address of word | 3F | 7-6 |
| PUSHL | Push long | DD | 6-8 |
| PUSHR | Push registers | BB | 7-2 |
| REI | Return from exception or interrupt | 02 | 12-13 |
| REMQUE | Remove from queue | 0F | 7-14 |

| MNEMONIC | INSTRUCTION | OPCODE | PAGE |
|---|---|---|---|
| RET | Return from called procedure | 04 | 8-24 |
| ROTL | Rotate long | 9C | 6-38 |
| RSB | Return from subroutine | 05 | 8-17 |
| SBWC | Subtract with carry | D9 | 6-25 |
| SCANC | Scan for character | 2A | 9-10 |
| SKPC | Skip character | 3B | 9-11 |
| SOBGEQ | Subtract one and branch on greater or equal | F4 | 8-13 |
| SOBGTR | Subtract one and branch on greater | F5 | 8-13 |
| SPANC | Span characters | 2B | 9-10 |
| SUBB2 | Subtract byte 2 operand | 82 | 6-22 |
| SUBB3 | Subtract byte 3 operand | 83 | 6-22 |
| SUBD2 | Subtract double 2 operand | 62 | 6-22 |
| SUBD3 | Subtract double 3 operand | 63 | 6-22 |
| SUBF2 | Subtract floating 2 operand | 42 | 6-22 |
| SUBF3 | Subtract floating 3 operand | 43 | 6-22 |
| SUBL2 | Subtract long 2 operand | C2 | 6-22 |
| SUBL3 | Subtract long 3 operand | C3 | 6-22 |
| SUBP4 | Subtract packed 4 operand | 22 | 10-8 |
| SUBP6 | Subtract packed 6 operand | 23 | 10-8 |
| SUBW2 | Subtract word 2 operand | A2 | 6-22 |
| SUBW3 | Subtract word 3 operand | A3 | 6-22 |
| SVPCTX | Save process context | 07 | 13-10 |
| TSTB | Test byte | 95 | 6-17 |
| TSTD | Test double | 73 | 6-17 |
| TSTF | Test float | 53 | 6-17 |
| TSTL | Test long | D5 | 6-17 |
| TSTW | Test word | B5 | 6-17 |
| XFC | Extended function call | FC | 13-6 |
| XORB2 | Exclusive OR byte 2 operand | 8C | 6-36 |
| XORB3 | Exclusive OR byte 3 operand | 8D | 6-36 |
| XORL2 | Exclusive OR long 2 operand | CC | 6-36 |
| XORL3 | Exclusive OR long 3 operand | CD | 6-36 |
| XORW2 | Exclusive OR word 2 operand | TC | 6-36 |
| XORW3 | Exclusive OR word 3 operand | AD | 6-36 |
| | *Reserved to DEC* | 57 | |
| | *Reserved to DEC* | 59 | |
| | *Reserved to DEC* | 5A | |
| | *Reserved to DEC* | 5B | |
| | *Reserved to DEC* | 5C | |
| | *Reserved to DEC* | 5D | |
| | *Reserved to DEC* | 5E | |
| | *Reserved to DEC* | 5F | |
| | *Reserved to DEC* | 77 | |
| ESCD | *Reserved to DEC* | FD | |
| ESCE | *Reserved to DEC* | FE | |
| ESCF | *Reserved to DEC* | FF | |

## B.2. OPCODE LISTING

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

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

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

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

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

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

# APPENDIX C

# PROCEDURE CALLING AND CONDITION HANDLING

## C.1 INTRODUCTION

This appendix specifies the software standard for use with the VAX-11 hardware procedure CALL mechanism. This standard is applicable to all externally CALLable interfaces in DIGITAL-supported standard system software. This standard is also applicable to inter-module CALLs to major VAX-11 components.

This standard does not apply to calls to internal (or local) routines. Within a single module, the language processor or programmer may use a variety of other linkage and argument-passing techniques.

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 actual arguments to the called procedure.
- function value return—the form and conventions on the use of the function value.
- register usage—which registers are preserved and who is responsible for preserving them.
- stack usage—rules governing the use of the stack.
- data types of arguments—the types of all arguments that can be passed.
- argument descriptor formats—how descriptors are passed for the more complex arguments.
- condition handling—how exceptional conditions are signalled and how they can be handled in a modular fashion.
- stack unwinding—how the current thread of execution can be aborted cleanly.

## C.2 GOALS

Goals for the VAX-11 procedure CALLing standard are:

1. The standard must be applicable to all of the inter-module CALLable interfaces in the VAX-11 software system. Specifically, the standard considers the requirements of BASIC, COBOL, FORTRAN, BLISS, assembler, and CALLs to the operating system. The needs of other languages that DIGITAL may support in the future must be noted. The standard should not include capabilities for lower-level components (e.g., BLISS, assembler, operating system) that cannot be invoked from the higher-level languages.

2. The calling program and called procedure can be written in different languages, including any of the above.
3. The procedure mechanism must be sufficiently economical in both space and time to be used and usable as the only calling mechanism within VAX-11.
4. The standard should contribute to the writing of error-free, modular, and maintainable software. Effective sharing and re-use of VAX-11 software modules is a significant goal.
5. The standard must allow the called procedure a variety of techniques for argument handling. The called procedure may (i) reference arguments indirectly through the argument list, (2) copy scalars and array addresses, (3) copy addresses of scalars and arrays.
6. Provide the programmer with some control over fixing, reporting, and flow of control on exceptions.
7. Provide subsystem and application writers with the ability to override system messages in order to give a more suitable application-oriented interface.
8. Add no space or time overhead to procedure calls and returns that do not establish handlers. Minimize time overhead for establishing handlers at the cost of increased time overhead when exceptions occur.

## C.3 CALLING SEQUENCE

At the option of the calling program, it invokes the called procedure using either the CALLG or CALLS instruction:

```
CALLG    Arglist.ab, Proc.ab
```

or

```
CALLS    Argcnt.rl, Pric.ab
```

CALLS pushes the argument count Argcnt.rl onto the stack as a longword and sets AP to the top of the stack. The complete sequence using CALLS is thus:

```
Push     Argn
. . .
Push     Arg1
CALLS    #n,Proc
```

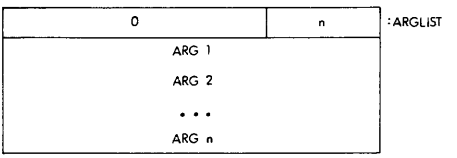
If the called procedure returns control to the calling procedure, control must return to the instruction immediately following the CALLG or CALLS instruction. Skip returns and GOTO returns are prohibited except during UNWIND.

The called procedure returns control to the calling procedure by executing the return instruction, RET.

## C.4 ARGUMENT LISTS

### C.4.1 Argument List Format

The format of the argument list is a sequence of longwords:




The argument count n is an unsigned byte contained in the first byte of the list. The high order 24 bits of the first longword are reserved to DIGITAL for future use and must be zero (MBZ). To access the argument count, the called procedure must ignore the reserved bits and pick up the count with the equivalent of a MOVZBL instruction.

Each Arg entry in the argument list is a 32-bit longword value. These 32-bit values may be:

1. An uninterpreted 32-bit value.
2. An address; typically a pointer to a scalar data item, an array, or a procedure.
3. An address of a descriptor; descriptors are discussed below.

The standard thus permits simple call-by-value, call-by-reference, call-by-descriptor, or combinations of these. Interpretation of each argument list entry depends upon agreement between the calling and called procedures.

A procedure having no arguments is CALLed with a list consisting of a 0 argument count longword. This is efficiently accompilshed by

```
CALLS     #0, Proc
```

A missing or null argument, for example CALL SUB (A, ,B), is represented on VAX-11 by an Arglist entry consisting of a longword 0. Some procedures allow trailing null arguments to be omitted, others require all arguments; refer to the procedure's specification for details.

The argument list must be treated as read-only data by the called procedure.

### C.4.2 Argument Lists And Higher-level Languages

Higher-level language functional notations for procedure CALLs are mapped into VAX-11 argument lists according to the following rules:

1. Actual arguments are mapped left-to-right to increasing argument list offsets. The left-most (first) actual argument corresponds to Arglist+4, the next to Arglist+8, etc.
2. Each actual argument position corresponds to a single VAX-11 argument list entry.

### C.4.2.1 Order Of Actual Argument Evaluation

Since most higher-level languages do not specify the order of evaluation (with respect to side effects) of actual arguments, those language processors may evaluate actual arguments in any convenient order.

In constructing an argument list on the stack, a language processor may evaluate arguments right-to-left and push their values on the stack. If call-by-reference is used, actual argument expressions can be evaluated left-to-right, with pointers to the expression values being pushed 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 actual arguments.

### C.4.2.2 Language Extensions For Argument Transmission

The VAX-11 procedure standard permits arguments to be transmitted by value, by reference, or by descriptor. Each language processor has a default set of argument mechanisms. Thus FORTRAN will pass scalars, arrays, and functions by reference, and will pass strings (CHARACTER) by descriptor. BASIC, however, will transmit both strings and arrays by descriptor.

A set of language extensions is defined to reconcile the different argument transmission techniques. Each language is extended to provide the user explicit control of argument transmission in the calling procedure.

Each language is augmented to provide the following compile-time intrinsic functions:

| | |
|---|---|
| %VAL (arg) | — Corresponding argument list entry is the actual 32-bit value of the argument arg, as defined in the language. |
| %REF (arg) | — Corresponding argument list entry is a pointer to the value of the argument arg, as defined in the language. |
| %DESCR (arg) | — Corresponding argument list entry is a pointer to a VAX-11 descriptor of the argument, as defined in this appendix and the language. |

These intrinsic functions can be used in the syntax of a procedure CALL to control generation of the actual argument list. For example:

```
CALL SUB1 (%VAL (123), %REF (X), %DESCR (A) )
```

The intrinsic functions are a necssary escape mechanism in permitting any procedure to be called by programs written in any higher-level language. Careful design of procedure packages will minimize the actual need to use these escape mechanisms.

## C.5 FUNCTION VALUE RETURN

A function value is returned in register R0 if representable in 32 bits, and 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 could not deal with them. If the function value cannot be represented

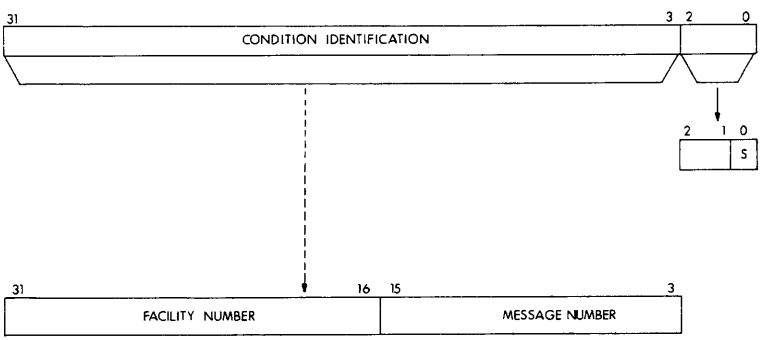
in 64 bits, the source language list of arguments and formals is shifted by one and the first formal in the argument list is reserved for the function value. One of the following mechanisms is used to return the function value:

1. If the maximum length of the function value is known, the calling procedure can allocate the required storage and pass a pointer to the function value storage as the first argument.
2. The calling procedure 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. This method requires a heap (non-stack) storage management mechanism.

Some procedures, such as operating system CALLs, return a success/fail value as a longword function value in R0. The value is used to encode the status refer to the next section.

## C.6 CONDITION VALUE

VAX-11 uses a standard means to report the success or failure of a called procedure and to describe an exception condition; see section C.11; when it occurs. This means is also used to identify system messages and to report program success or failure for command language testing. 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:




condition identification identifies the condition uniquely on a system-wide basis.

facility identifies the software component generating the condition value. Bit 31 is set for customer facilities and clear for DIGITAL facilities.

message number is a status identification; it 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 is the severity code as follows:

severity <0> is set for success (logical true)

severity <2:1> distinguishes degrees of success or failure.

Thus, the field <2:0> can be considered as a number

| | |
|---|---|
| STS$K_WARNING | 0 = warning |
| STS$K_SUCCESS | 1 = success |
| STS$K_ERROR | 2 = error |
| | 3 = reserved to DIGITAL |
| STS$K_SEVERE | 4 = severe_error |
| | 5, 6, 7 reserved to DIGITAL |

Software symbols are defined for these fields as follows:

| MNEMONIC | VALUE | MEANING | FIELD |
|---|---|---|---|
| STS$V_COND_ID | 3 | position of 31:3 | condition identification |
| STS$S_COND_ID | 29 | size of 31:3 | |
| STS$M_COND_ID | mask | mask for 31:3 | |
| STS$W_FAC_NO | 2 | word for 31:16 | facility number |
| STS$V_CUST_DEF | 31 | position for 31 | customer facility |
| STS$S_CUST_DEF | 1 | size for 31 | |
| STS$M_CUST_DEF | 1 @ 31 | mask for 31 | |
| STS$V_MSG_NO | 3 | position of 15:3 | message number |
| STS$S_MSG_NO | 13 | size of 15:3 | |
| STS$M_MSG_NO | mask | mask for 15:3 | |
| STS$V_FAC_SP | 15 | position of 15 | facility specific |
| STS$S_FAC_SP | 1 | size for 15 | |
| STS$M_FTC_SP | 1 @ 15 | mask for 15 | |
| STS$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 | |

### C.6.1 Interpretation of Severity Codes

A severity code of 1 indicates that the procedure generating the condition value was completed successfully, i.e., as expected.

A severity code of 0 indicates a warning. This code is used whenever a procedure produces output but the result might not be what the user expected, e.g., a compiler has modified a source program.

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 4 indicates that a severe error has 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 routine typically performs a low bit test, so it treats warnings, errors, and severity_errors as failures. If the condition value is signalled (see section C.11), 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 exists, the command interpreter by default treats errors and severe_errors as the basis for stopping the job, and warnings and successes as the basis for continuing.

Thus, the following table summarizes the default interpretation of condition values:

| SEVERITY | ROUTINE | SIGNAL | DEFAULT AT PROGRAM EXIT |
|---|---|---|---|
| success | normal | continue | continue |
| warning | failure | continue | continue |
| error | failure | continue | stop job |
| severe_error | failure | exit | stop job |

Unless there is a good basis for another choice, a routine should use either success or severe_error as its severity for each condition value.

### C.6.2 Use of Condition Values

Software components produced by DIGITAL for VAX-11 return condition values when they complete execution. When a severity code of warning, error, or severe_error has been generated, the status code describes the nature of the problem. This value may be tested to change the flow of control of a procedure and/or be used to generate a message. User procedures may also generate condition values to be examined by other procedures and by the command language interpreter. User-generated values should set bit 31 and bit 15 so that these condition values will not conflict with values generated by DIGITAL.

## C.7 REGISTER USAGE

The following registers have defined uses:

PC —program counter

SP —stack pointer

FP —current stack frame pointer. Must always point at current frame; no modification permitted within a procedure body.

AP —At the instant of CALL, AP must point to a valid argument list. A parameterless procedure points to an argument list consisting of a single longword containing the value 0.

R0, R1 —Function value return registers. These registers are not preserved by any called procedure. They are available as "free temporaries" to any called procedure.

All other registers (R2, R3, . . . , R10, R11, and AP, FP, SP, PC) are preserved across procedure calls. The called procedure may use any of these provided that 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. If JSB routines are used, they must not save any registers not already saved by the entry mask mechanism of the calling program.

## C.8 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 C.11. Any other use of FP at any time within a procedure is prohibited.

The content of the stack located at higher addresses than the mask/PSW longword belongs to the calling program; it should not be read or written by the called procedure, except as specified in the argument list. The content of the stack located at lower addresses than (SP) belongs to interrupt and exception routines; it must be assumed to be continually and unpredictably modified.

Local storage is allocated by the called procedure 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.

## C.9 ARGUMENT DATA TYPES

The following encoding is used for atomic data elements:

| MNEMONIC | CODE | DESCRIPTION |
|---|---|---|
| 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_V | 1 | Bit. Ordinarily a bit string; see discussion of descriptors. |
| 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$K_DTYPE_B | 6 | Byte Integer. 8-bit signed 2's-complement integer. |
| DSC$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$K_DTYPE_F | 10 | Single-precision Floating. 32-bit VAX-11 floating point. |
| DSC$K_DTYPE_D | 11 | Double-precision Floating. 64-bit VAX-11 floating point. |
| DSC$K_DTYPE_FC | 12 | Complex. Ordered pair of single-precision floating quantities, representing a complex number. The lower addressed quantity represents the real part, the higher addressed represents the imaginary part. |
| DSC$K_DTYPE_DC | 13 | Double-precision Complex. Ordered pair of double-precision floating point quantities, representing a complex number. The lower addressed quantity represents the real part, the higher addressed represents the imaginary part. |

The following string types are ordinarily described by a string descriptor. The data type codes below occur in those descriptors:

| MNEMONIC | CODE | DESCRIPTION |
|---|---|---|
| DSC$K_DTYPE_T | 14 | ASCII text string. A sequence of 8-bit ACSII characters. |
| DSC$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. |
| DSC$K_DTYPE_ZI | 22 | Sequence of instructions. |
| DSC$K_DTYPE_ZEM | 23 | Procedure entry mask. |

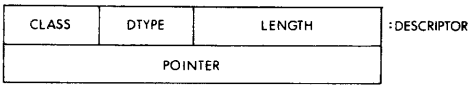
The following type codes are reserved for future use:

| | |
|---|---|
| 24-191 | reserved to DIGITAL |
| 192-255 | reserved to CSS and customers |

## C.10 ARGUMENT DESCRIPTORS

A uniform descriptor mechanism is defined for use by all procedures that conform to this standard. Descriptors are uniformly typed and the mech-

anism is extensible. Each class of descriptor consists of at least 2 longwords in the following format:

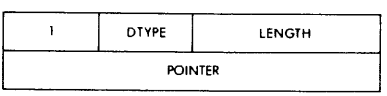



| | |
|---|---|
| DSC$W_LENGTH <0, 15:0> | A one-word field specific to the descriptor class; typically a 16-bit (unsigned) length. |
| DSC$B_DTYPE <0, 23:16> | A one-byte atomic data type code (see C.9). |
| DSC$B_CLASS <0, 31:24> | A one-byte descriptor class code (see below). |
| DSC$T_POINTER <1, 31:0> | A longword pointing to 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 used directly. This gives a word length, so the class and type are placed as bytes in the rest of that longword. Class 0 is unspecified and hence no more than the above information can be assumed.

### C.10.1 Scalar, String Descriptor (DSC$K_CLASS_S)

A single descriptor form is used for scalar data and simple strings.




| | |
|---|---|
| DSC$W_LENGTH | Length of data item in bytes, unless DTYPE EQLU 1 (Bit) or 21 (Packed Decimal). Length of data item is in bits for bit string. Length of data item is in digits (nibbles-1) for packed string. |
| DSC$B_DTYPE | |
| DSC$B_CLASS | 1 = DSC$K_CLASS_S |
| DSC$A_POINTER | Address of first byte of data storage |

### C.10.2 Dynamic String Descriptor (DSC$K_CLASS_D)

Reserved to DIGITAL.

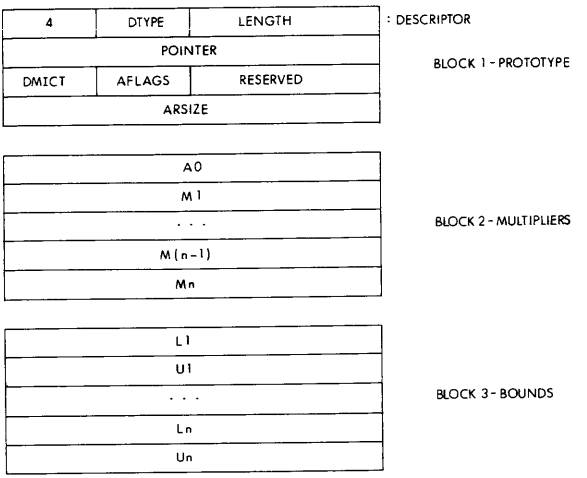
### C.10.3 Varying String Descriptor (DSC$K_CLASS_V)

Reserved to DIGITAL.

### C.10.4 Array Descriptor (DSC$K_CLASS_A)

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 then so is the second. A complete array descriptor has the form:

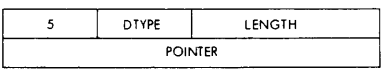



| | |
|---|---|
| DSC$W_LENGTH | Data element size (in bytes unless DTYPE EQLU 1 or 21) |
| DSC$B_DTYPE<br>DSC$B_CLASS | 4 = DSC$K_CLASS_A |
| DSC$A_POINTER | Address of first actual byte of data storage. |
| Reserved<br><2, 15:0> | Reserved for future use (MBZ). |
| DSC$B_AFLAGS<br><2, 23:16> | Array flag bits. |
| Reserved<br><2, 20:16> | MBZ |
| DSC$V_FL_COLUMN<br><2, 21> | If set, the elements of the array are stored by columns (FORTRAN). Otherwise the elements are stored by rows. |
| DSC$V_FL_COEFF<br><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<br><2, 23> | If set, the bounds information in Block 3 is present. Requires that DSC$V_FL_COEFF be set. |
| DSC$B DIMCT<br><2, 31:24> | Number of dimensions |
| DSC$L_ARSIZE<br><3, 31:0> | Total size of array<br>(in bytes unless DTYPE EQLU 1 or 21). |

| | |
|---|---|
| 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 0-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 of i'th dimension. |
| DSC$L_Ui <4+n+2*i, 31:0> | Upper bound of i'th dimension. |

### C.10.5 Procedure Descriptor (DSC$K_CLASS_P)

The descriptor for a procedure specifies its entry address and function value data type, if any.



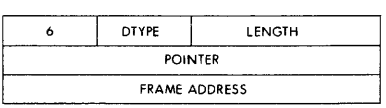

| | |
|---|---|
| DSC$W_LENGTH | Length associated with the function value. |
| DSC$B_DTYPE | Function value data type. |
| DSC$B_CLASS | 5 = DSK$K_CLASS_P |
| DSC$A_POINTER | Address of entry mask to routine. |

Procedures return values in R0 or R0 and R1 as follows:

1. If a scalar, then the value is in R0 or R0 and R1. The type and length are specified as DSC$B_DTYPE and DSC$W_LENGTH in the procedure descriptor.
2. If not a scalar (i.e., if an array, a string, a procedure, etc.), then no function value may be returned. Instead, the argument expressed as a function value is instead passed as the first argument and the other arguments are shifted down by one.

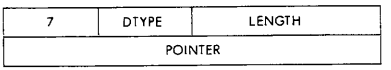
### C.10.6 Procedure Incarnation Descriptor (DSC$K_CLASS_PI)

The descriptor for a procedure incarnation is the same as a procedure descriptor with the addition of its call frame address. This is used to refer to a specific incarnation of a procedure.




| | |
|---|---|
| DSC$W_LENGTH | Length associated with the function value |
| DSC$B_DTYPE | Function value data type. |
| DSC$B_CLASS | 6 = DSC$K_CLASS_PI |
| DSC$A_POINTER | Address of entry mask to routine. |
| DSC$A_FRAME <2, 31:0> | Address of frame of this incarnation. |

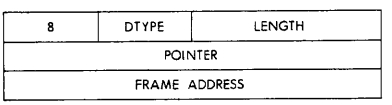
### C.10.7 Label Descriptor (DSC$K_CLASS_J)




| | |
|---|---|
| DSC$W_LENGTH | Not used; MBZ. |
| DSC$B_DTYPE | Not used; MBZ. |
| DSC$B_CLASS | 7 = DSC$K_CLASS_J |
| DSC$A_POINTER | Address of label to jump to. |

### C.10.8 Label Incarnation Descriptor (DSC$K_CLASS_JI)

The descriptor for a label incarnation is the same as a label descriptor with the addition of its procedure incarnation's call frame address. This is used to refer to a label within a specific incarnation of a procedure.




| | |
|---|---|
| DSC$W_LENGTH | Not used; MBZ. |
| DSC$B_DTYPE | Not used; MBZ. |
| DSC$B_CLASS | 8 = DSC$K_CLASS_JI. |
| DSC$A_POINTER | Address of label to jump to. |
| DSC$A_FRAME | Address of frame of this incarnation. |

<2, 31:0>

### C.10.9 Reserved Descriptors

Descriptor classes 9-191 are reserved to DIGITAL. Classes 192 through 255 are reserved to CSS and customers.

## C.11 VAX-11 CONDITIONS

A condition is a hardware generated synchronous exception or the occurrence of a software event that the program wishes to process in a manner analogous to a hardware exception. Floating overflow trap, memory access violation exception, and the reserved operation exception are examples of hardware generated conditions. The occurrence of an output conversion error, an end-of-file or the filling of an output buffer are examples of software detected events that might be treated as conditions.

Depending on the condition and on the program, there are four types of actions that might be taken when a condition occurs.

1. Ignore the condition. For example, if an underflow occurs in a VAX-11 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 where the condition occurred. For example, the end of a buffer is reached while writing a series of data items. The special action is to start a new buffer.

3. Terminate the operation and branch from the sequential flow of control. For example, 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, the floating divide by zero exception condition occurs. The program exits, possibly after writing an appropriate error message.

When an unusual event or error occurs in a called procedure, the procedure can return a condition value to the caller which indicates what has happened, see section C.6. The caller then tests the condition value and 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 in the case of certain software generated events, it is more convenient to handle the condition as soon as it is detected rather than to program explicit tests.

### C.11.1 Condition Handlers

For the primary purpose of handling hardware-detected exceptions, the VAX/VMS system supplies a mechanism for the programmer to specify a condition handler function to be called when an exception condition occurs. This mechanism may also be used for software detected exceptions.

An active procedure may establish a condition handler to be associated with it. The presence of a condition handler is indicated by a non-zero address in a 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 same format and interpretation as that returned as a function value, see Section C.6. All hardware exceptions are signalled.

When a condition is signalled 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.

As an example, consider a procedure that wishes to keep track of the occurrence of the floating underflow exception. The procedure can establish a condition handler to examine the condition value passed when the handler is invoked and, when the condition is underflow, log the exception. When the floating underflow exception occurs, the condition handler will be entered. After logging the condition, the handler can 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 might be associated with the program's main procedure. When the condition is signalled, successive stack frames will be

searched until the stack frame for the main procedure is found and then 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.

### C.11.2 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 signalled 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 C.6, 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 may:

1. Return to the instruction immediately following the signal. Note that it is not always possible to make such a return.
2. Resignal the condition or a modified condition value. The search for another condition handler will then begin with the immediately preceeding stack frame.
3. Signal a different condition. Refer to section C.14, Multiple Active Signals.
4. Unwind the stack. Refer to Section C.13.4, Request to Unwind.

## C.12 OPERATIONS INVOLVING CONDITION HANDLERS

The VAX-11 system provides facilities to support the condition handling mechanism. The functions provided are:

1. Establish a condition handler. A condition handler is associated with the current procedure by placing the handler's address in the current procedure activation's stack frome.
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 exceptions floating underflow, integer overflow, and decimal overflow may be enabled or disabled by the software. No signal occurs when the exception is disabled; see Chapter 12.
4. Signal a condition. Signalling 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 pre-signal frames 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; see section C.13.4. Unwind allows a procedure to perform application-specific cleanup operations before exiting.

### C.12.1 Establish A Condition Handler

Each procedure activation has a condition handler potentially attached to it via a longword in its stack frame. Initially, the longword 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, the VAX/VMS operating system provides two exception vectors at each access mode. 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, whether or not handled. 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:

```
MOVAL     handler_entry_point, 0(FP)
```

### C.12.2 Revert Condition Handler

The revert handler operation deletes the condition handler associated with the procedure activation. This is done by clearing the handler address in the stack frame.

The code to revert a handler is:

```
CLRL      0(FP)
```

### C.12.3 Signal a Condition

The signal operation is the method used for indicating that an exception condition has occurred. When a program wishes to issue a message and potentially continue execution after handling the condition, it calls the standard procedure:

```
CALL LIB$SIGNAL   (condition_value, arg_list . . .)
```

When a program wishes to issue a message and never continue, it calls the standard procedure:

```
CALL LIB$STOP   (condition_value, arg_list . . .)
```

where in both cases condition_value indicates the condition that is being signalled; see section C.6. The arguments arg_list describe the details of the exception. These are the same arguments used to issue a system message. Note that unlike most CALLs, LIB$SIGNAL preserves R0 and R1 as well as the other registers. This allows a debugger to display the entire state of the process at the time of the exception. It also allows signals to be placed in assembly language code without changing the register usage. This is useful for debugging checks and statistical gathering. Hardware and system service exceptions behave as though they were a call to LIB$SIGNAL.

The signal procedure examines the two exception vectors and then up to 64K previous stack frames. 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 are a pair of address locations per access mode.

A frame before the call by the system command interpreter to the main program establishes a default condition handler that issues system messages. The default condition handler uses condition_value to get the message and then uses the arg_list to format and output the message. If the condition_value <2:0> is not severe_error (i.e., 4) the default condition handler returns with SSS_CONTINUE; if the severity is severe_ error the default handler exits the program image with the condition value as the final image status.

The stack search terminates when the old FP is 0 or is not accessible or 64K frames have been examined. If no condition handler is found, or all handlers returned with the SS$_RESIGNAL, then SIGNAL issues a message that no handler was found and then issues the SIGNALed message and exits.

If a handler returns SS$_CONTINUE, and LIB$STOP was not called, then control returns to the signaler. Otherwise LIB$STOP issues a message that there was an attempt to continue from a non-continuable exception and exits with the condition value as the final image status.

All combinations of interaction between condition handler actions, the default condition handler, the type of signal, and the call to signal or stop are detailed in the following table.

| | signaled condition <2:0> | default handler gets control | handler specifies continue | handler specifies UNWIND | no handler is found (stack bad) |
|---|---|---|---|---|---|
| call to: SIGNAL | <4 | condition message RET | RET | UNWIND | "no handler found" condition message EXIT |
| | =4 | condition message EXIT | RET | UNWIND | "no handler found" condition message EXIT |
| STOP | <4 | condition message "can't continue" EXIT | "can't continue" EXIT | UNWIND | "no handler found" condition message EXIT |
| | =4 | condition message EXIT | "can't continue" EXIT | UNWIND | "no handler found" condition message EXIT |

condition message is the standard message for the condition value

"no handler found" is a standard message that indicates that no condition handler was found (i.e., that the stack is bad). The message distinguishes between no handler (old FP = 0 or too many frames) and access violation (old FP = junk).

"can't continue" is a standard message that indicates an attempt to continue from a call to LIB$STOP.

## C.13 PROPERTIES OF CONDITION HANDLERS

### C.13.1 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)

where 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 reference 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. Since bits 2:0 of the condition_value indicate severity and do not indicate which condition is being signalled, the handler should examine only the condition identification ,i.e., condition_value <31:3>. The setting of bits <2:0> 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 reference the elements of the signal vectors.

Mechanism_args is a vector of five longwords

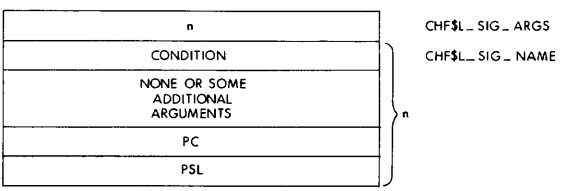
| | |
|---|---|
| 4 | CHF$L_MCH_ARGS |
| FRAME | CHF$L_MCH_FRAME |
| DEPTH | CHF$L_MCH_DEPTH |
| R0 | CHF$L_MCH_SAVR0 |
| R1 | CHF$L_MCH_SAVR1 |

CHF$L_MCH_ARGS
CHF$L_MCH_FRAME
CHF$L_MCH_DEPTH
CHF$L_MCH_SAVR0
CHF$L_MCH_SAVR1

Frame is the contents of FP in the establisher's context. This can be used as a base to reference the local storage of the establishers if the restrictions in section C.13.2 are met. Depth is a positive counter of the

number of procedure activation stack frames above the signal that the condition handler was established. Depth has the value 0 for an exception handed by the activation invoking the exception (i.e., containing the instruction causing the hardware exception or calling LIB$SIGNAL). Depth has positive values for procedure invocations calling the one having the execution (1 for the immediate caller, etc.). If a system service gives an exception, the immediate caller of the service gets notified at depth = 1. Depth has value -2 when the condition handler is established by the primary exception vector and -1 when it is established by the secondary vector. R0 and R1 are the values of these registers at the time of the call to LIB$SIGNAL.

For hardware detected exceptions, the condition_value indicates which exception vector was taken and the next 0 or several longwords are the additional parameters as specified in Chapter 12. The remaining two longwords are the PC and PSL:




### C.13.2 Use Of Memory

In order not to impact compiler optimization, a handler and anything it calls is restricted to referencing only explicitly passed arguments. They cannot reference COMMON or other external storage and they cannot reference local storage in the procedure that established the handler. Compilers relaxing this rule must ensure that any variables referenced by the handler are always kept in memory (VOLATILE) and have a full lifetime.

### C.13.3 Returning From a Condition Handler

Condition handlers are invoked by the standard system procedure that processes signals, therefore the return from the condition handler is to this procedure.

If the handler wishes execution to continue from the instruction following the signal, it must return with the function value SS$_CONTINUE ("true," i.e., with bit 0 set). If, however, the condition was signalled with a call to LIB$STOP, the image will exit. If it wishes the condition to be resignalled, the condition handler returns with the function value SS$_RESIGNAL ("false," i.e., with bit 0 clear3. If the handler wants to alter the severity of the signal, it modifies the low three bits of condition_value and resignals. If the handler wants to unwind, it calls SYS$-UNWIND and then returns; see Section C.13.4. In this case the handler function value is ignored.

### C.13.4 Request To Unwind

If the handler decides to unwind, the handler or any procedure it calls performs:

```
success = SYS$UNWIND
                ([dept = {handler depth} + 1],
                 [new_PC = {return PC}] )
```

The argument depth specifies how many pre-signal frames to remove. If depth is LEQ 0 then nothing is to be unwound. The default is to return from the establisher of the handler. To unwind to the establisher, the depth from the call to the handler should be specified. When the handler is found at depth 0, the equivalent of UNWIND is to alter the PC in the handler mechanism_args.

The argument new_PC specifies the location to receive control when the unwind is complete. The function value is a standard success code (SS$_NORMAL), or indicates the failure "no signal active" (SS$NO-SIGNAL), "already unwinding" (SS$_UNDINDING), or "insufficient frames for depth" (SS$_INSFRAME).

The unwind will happen 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, then the current handler will be recalled with this unwind exception.

The call to the handler is of the same form as described above with the following values:

```
signal_args
    1
    condition_value = SS$_UNWIND

mechanism_args
    4
    frame       establisher's frame
    depth       0 (i.e., unwinding self)
    R0          R0 that unwind will restore
    R1          R1 that unwind will restore
```

After each handler has been 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. If the handler wants to specify the value of the top level "function" being unwound, it should modify R0 and R1 in the mechanism vector because they will be restored from the mechanism argument vector at the end of the unwind.

Depending on the arguments to SYS$UNWIND, the unwinding operation will be terminated as follows:

1. SYS$UNWIND (0, 0)—unwind to the establisher's caller with establisher function value from R0 and R1 in the mechanism vector.
2. SYS$UNWIND (depth, 0)—unwind to the establisher at the point of the call that resulted in the exception. The callee's function value is taken from R0 and R1 in the mechanism vector.
3. SYS$UNWIND (depth, location)—unwind to a specified activation and transfer to a specified location with R0 and R1 from the mechanism vector.

### C.13.5 Signaller's Registers

Because the handler is called, and may 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. In order to find the registers R2 through FP, a scan up the stack frames must be performed. During the scan, the last frame found to save a register contains that register's contents. If no frame back to the call to the handler saved the register, then 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 |
| PC, PSL: | at end of signal_args |

## C.14 MULTIPLE ACTIVE SIGNALS

A signal is said to be active until the signaler gets control again or is unwound. It is possible for a signal to occur while a condition handler or a procedure it has called is executing. Consider the following example. For each procedure (A, B, C, . . .) let the condition handler it establishes be (Ah, Bh, Ch, . . .). If A calls B calls C which signals "S" and Ch resignals, then Bh gets control. If Bh calls X calls Y which signals "T" the stack is:

```
<signal T>
    Y
    X
    Bh
<signal S>
    C
    B
    A
```

which was programmed:

```
A
    B - - - - - - - - - - - - - - - - - -> Bh
        C                                    X
    <signal S>                                 Y
                                             <signal T>
```

The desired order to search for handlers is Yh, Vh, <Bh>h. Ah. Note that Ch should not be called because it is a structural descendant of B. Bh should not be called again because that would require it to be recursive. If it were recursive, then handlers could not be coded in non-recursive languages such as FORTRAN. Instead Bh can establish itself or another procedure as its handler (Bhh).

To implement this, the following algorithm is used. As usual, 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.

For proper hierarchical operation, an exception occurring during execution of a condition handler established in an exception vector should be handled by that handler rather than propagating up the activation stack. This is the vectored condition handler's responsibility. It is most easily accomplished by the vectored handler establishing a catch-all handler.

## APPENDIX D

# PROGRAMMING EXAMPLES

### D.1 PURPOSE

The purpose of the programming examples is to illustrate VAX-11 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.

### D.2 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-11 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-11 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     ^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
```

Line1 contains an entry mask so that registers R2 and R3 will be saved by the CALL instruction which calls the subroutine. By conveniton, 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.

Lines 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 works 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 |

## D.3 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       ^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        #^FI. 0, R0                 ;if negative, get
            DECL        R2                          ;positive reduction
1$:         BICB2       #^C7, R2                    ;mask to 8 octants
            CASEB       R2, #1, #6                  ;branch to each octant
2$:         .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 handler
            .WORD       ...
            ...
```

## D.4 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$FLOAT        4                 ;float sign, move 4 digits
          EO$END_FLOAT                      ;end floating sign
          EO$MOVE         1                 ;move one digit
          EO$INSERT       ^A/./             ;insert period
          EO$MOVE         3                 ;move three fractional digits
          EO$END                            ;end of pattern

FOUT: :
          .WORD           ^XC03C            ;save R2-R5, enable overflows
          SUBL2           #8, SP            ;make room on stack
          MULF3           #^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
```

## D.5 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,@A,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.

1. 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
```

2. PICTURE +$99,999.99

```
   MICRO:       EO$ADJUST_INPUT     7
                EO$LOAD_PLUS        '+
                EO$STORE_SIGN
                EO$SET_SIGNIF
                EO$INSERT           '$
                EO$MOVE             2
                EO$INSERT           ',
                EO$MOVE             3
                EO$INSERT           '.
                EO$MOVE             2
                EO$LOAD_FILL        '+
                EO$REPLACE_SIGN     11
                EO$END
```

3. PICTURE ZZ,ZZZ.ZZ

```
   MICRO:       EO$ADJUST_INPUT     7
                EO$MOVE             2
                EO$INSERT           ',
                EO$MOVE             3
                EO$SET_SIGNIF
                EO$INSERT           '.
                EO$MOVE             2
                EO$BLANK_ZERO       3
                EO$END
```

4. 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
```

5. 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
```

6. 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
```

7. 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
```

8. 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
```

## D.6 FORTRAN STATEMENT EVALUATION

**FORTRAN**
**Assembly**
**Code:** J = A*K + B(I)

```
MOVL I, R1      ;Move I to R1
CVTLF K, R0     ;Convert integer K to floating
                 point
MULF2 A, R0     ;Multiply A*K and store in R0
ADDF2 B
[R1], R0        ;Add B indexed by R1 to 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 is 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.

## D.7 VARIABLE LENGTH FIELD

**PL1**
**Assembly**
**Code:**

```
DECLARE A (1:10) BIT (5)   ;Vector A, elements 1-10,
                           ;5 bit field
A(I) = A(I) + 1            ;Increment Ith element of
                           ;A and store in A
```

**Machine**
**Code:**

```
INDEX I, #1, #10, #5,
#-5, R0          ;Calculate index
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, a 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.

## D.8 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:**

```
START:    MOVW #1, L
          ACBW #10, #2, L, START
```

## D.9 LOOPS

**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
```

## D.10 CHARACTER STRING

Translation Portion

Character String A

| | 1 | 67 | 97 | 26 | 27. . . |
|---|---|---|---|---|---|
| ASCII | CTRL/A | C | a | CTRL/Z | ESC |

Resulting string B

| | 1 | 67 | 65 | 26 |
|---|---|---|---|---|
| ASCII | CTRL/A | C | A | CTRL/Z |

Look up value in table plus number in string A; store result in string B.

When output is 27, instruction stops and condition code V is set.

The instruction can terminate by:

1. Input string running out
2. Output string running out, or
3. Encountering escape sequence.

If input is longer than output, no C bit is generated. If output is longer than input, C bit is set (normal termination).

# APPENDIX E

# OPERAND SPECIFIER NOTATION

## E.1 OPERAND SPECIFIERS

Operand specifiers are described in the following way:

<name><access type><data type>

where:

1. Name is a suggestive name for the operand in the context of the instruction. The name is often abbreviated.
2. 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.
3. Data type is a letter denoting the data type of the operand:
   - b — byte
   - d — double floating
   - f — floating
   - l — longword
   - q — quadword
   - w — word
   - x — first data type specified by instruction
   - y — second data type specified by instruction

## E.2 OPERATION DESCRIPTION NOTATION

The operation of each instruction is given as a sequence of control and assignment statements in an ALGOL-like syntax. No attempt is made to define the syntax formally; it is assumed to be familiar to the reader.

+ addition

— subtraction, unary minus

* multiplication

/ division (quotient only)

** exponentiation

' concatenation

← is replaced by

= is defined as

Rn or R[n] contents of register Rn

PC, SP, CF, or AP the contents of register R15, R14, R13, or R12 respectively

PSW the contents of the processor status word

PSL the contents of the processor status long word

(x) contents of memory location whose address is x

(x) + contents of memory location whose address is x; x incremented by the size of operand referenced at x

— (x) x decremented by size of operand to be referenced at x; contents of memory location whose address is x

<x:> 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

{ } — arithmetic parentheses used to indicate precedence

AND logical AND

OR logical OR

XOR logical XOR

NOT logical (ones) 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 sign 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

MINU (x, y) minimum unsigned of x and y

The following conventions are used:

1. 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.
2. No operator precedence is assumed, other than that replacement ( ← ) has the lowest precedence. Precedence is indicated explicitly by { }.
3. All arithmetic, logical, and relational operators are defined in the context of their operands. 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.
4. 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.
5. Condition codes are in general affected on the value of actual stored results, not on "true" results (which might be generated internally to greater precision). Thus, for example, 2 positive integers can be added together and the sum stored, because of overflow, as a negative value. The condition codes will indicate a negative value even though the "true" result is clearly positive.

# GLOSSARY

**abort** An exception that occurs in the middle of an instruction and potentially leaves the registers and memory in an indeterminate state, 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** In absolute mode addressing, the PC is used as the register in autoincrement deferred mode. The PC contains the address of the location containing the actual operand.

**access mode** 1. Any of the four processor access modes in which software executes. Processor access modes are, in order from most to least privileged and protected: kernel (mode 0), executive (mode 1), supervisor (mode 2), and user (mode 3). When the processor is in kernel mode, the executing software has complete control of, and responsibility for, the system. When the processor is in any other mode, the processor is inhibited from executing privileged instructions. The Processor Status Longword contains the current access mode field. The operating system uses access modes to define protection levels for software executing in the context of a process. For example, the executive runs in kernel and executive mode and is most protected. The command interpreter is less protected and runs in supervisor mode. The debugger runs in user mode and is not more protected than normal user programs. 2. See record access mode.

**access type** 1. The way in which the processor accesses instruction operands. Access types are: read, write, modify, address, and branch. 2. The way in which a procedure accesses its arguments.

**access violation** An attempt to reference an address that is not mapped into virtual memory or an attempt to reference an address that is not accessible by the current access mode.

**address** A number used by the operating system and user software to identify a storage location. See also virtual address and physical address.

**address access type** The specified operand of an instruction is not directly accessed by the instruction. The address of the specified operand is the actual instruction operand. The context of the address calculation is given by the data type of the operand.

**addressing mode** The way in which an operand is specified; for example, the way in which the effective address of an instruction operand is calculated using the general registers. The basic general register addressing modes are called: register, register deferred, autoincrement, autoincrement deferred, autodecrement, displacement, and displacement deferred. In addition, there are six indexed addressing modes using two general registers, and literal mode addressing. The PC addressing modes are called: immediate (for register deferred mode using the PC), absolute (for autoincrement deferred mode using the PC), and branch.

**address space** The set of all possible addresses available to a process. Virtual address space refers to the set of all possible virtual addresses. Physical address space refers to the set of all possible physical addresses sent out on the SBI.

**alphanumeric character** An upper or lower case letter (A-Z, a-z), a dollar sign ($), an underscore (_), or a decimal digit (0-9).

**American Standard Code for Information Interchange (ASCII)** A set of 8-bit binary numbers representing the alphabet, punctuation, numerals, and other special symbols used in text representation and communications protocol.

**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 indexed mode** An indexed addressing mode in which the base operand specifier uses autodecrement mode addressing.

**autodecrement mode** In autodecrement mode addressing, the contents of the selected register are decremented, and the result is used as the address of the actual operand for the instruction. The contents of the register are decremented according to the data type context of the register: 1 for byte, 2 for word, 4 for longword and floating, 8 for quadword and double floating.

**autoincrement deferred indexed mode** An indexed addressing mode in which the base operand specifier uses autoincrement deferred mode addressing.

**autoincrement deferred mode** In autoincrement deferred mode addressing, the specified register contains the address of a longword which contains the address of the actual operand. The contents of the register are incremented by 4 (the number of bytes in a longword). If the PC is used as the register, this mode is called absolute mode.

**autoincrement indexed mode** An indexed addressing mode in which the base operand specifier uses autoincrement mode addressing.

**autoincrement mode** In autoincrement mode addressing, the contents of the specified register are used as the address of the operand, then the contents of the register are incremented by the size of the operand.

**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 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 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 addressing mode, the instruction operand specifier is a signed byte or word displacement. The displacement is added to the contents of the updated PC (which is the address of the first byte beyond the displacement), and the result is the branch address.

**byte** A byte is eight contiguous bits starting on an addressable byte boundary. Bits are numbered from the right, 0 through 7, with bit 0 the low-order bit. When interpreted arithmetically, a byte is a two's complement integer with significance increasing from bits 0 through 6. Bit 7 is the sign bit. The value of

the signed integer is in the range -128 to 127 decimal. When interpreted as an unsigned integer, significance increases from bits 0 through 7 and the value of the unsigned integer is in the range 0 to 255 decimal. A byte can be used to store one ASCII character.

**cache memory** A small, high-speed memory placed between slower main memory and the processor. A cache increases effective memory transfer rates and processor speed. It contains copies of data recently used by the processor, and fetches several bytes of data from memory in anticipation that the processor will access the next sequential series of bytes.

**call frame** See stack frame.

**call instructions** The processor instructions CALLG (Call Procedure with General Argument List) and CALLS (Call Procedure with Stack Argument List).

**call stack** The stack, and conventional stack structure, used during a procedure call. Each access mode of each process context has one call stack, and interrupt service context has one call stack.

**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.

**compatibility mode** A mode of execution that enables the central processor to execute non-privileged PDP-11 instructions. The operating system supports compatibility mode execution by providing an RSX-11M programming environment for an RSX-11M task image. The operating system compatibility mode procedures reside in the control region of the process executing a compatibility mode image. The procedures intercept calls to the RSX-11M executive and convert them to the appropriate operating system functions.

**condition** An exception condition detected and declared by software. For example, see failure exception mode.

**condition codes** Four bits in the Processor Status Word that indicate the results of previously executed instructions.

**condition handler** A procedure that a process wants the system to execute when an exception condition occurs. When an exception condition occurs, the operating system searches for a condition handler and, if found, initiates the handler immediately. The condition handler may perform some action to change the situation that caused the exception condition and continue execution for the process that incurred the exception condition. Condition handlers execute in the context of the process at the access mode of the code that incurred the exception condition.

**condition value** A 32-bit quantity that uniquely identifies an exception condition.

**context indexing** The ability to index through a data structure automatically because the size of the data type is known and used to determine the offset factor.

**context switching** Interrupting the activity in progress and switching to another activity. Context switching occurs as one process after another is scheduled for execution. The operating system saves the interrupted process' hardware context in its hardware process control block (PCB) using the Save Process Context instruction, loads another process' hardware PCB into the hardware context using the Load Process Context instruction, scheduling that process for execution.

**console** The manual control unit integrated into the central processor. The console includes an LSI-11 microprocessor and a serial line interface connected to a hard copy terminal. It enables the operator to start and stop the system, monitor system operation, and run diagnostics.

**console terminal** The hard copy terminal connected to the central processor console.

**control region** The highest-addressed half of per-process space (the P1 region). Control region virtual addresses refer to the process-related information used by the system to control the process, such as: the kernel, executive, and supervisor stacks, the permanent I/O channels, exception vectors, and dynamically used system procedures (such as the command interpreter and RSX-11M programming environment compatibility mode procedures). The user stack is also normally found in the control region, although it can be relocated elsewhere.

**Control Region Base Register (P1BR)** The processor register, or its equivalent in a hardware process control block, that contains the base virtual address of a process control region page table.

**Control Region Length Register (P1LR)** The processor register, or its equivalent in a hardware process control block, that contains the number of nonexistent page table entries for virtual pages in a process control region.

**counted string** A data structure consisting of a byte-sized length followed by the string. Although a counted string is not used as a procedure argument, it is a convenient representation in memory.

**current access mode** The processor access mode of the currently executing software. The Current Mode field of the Processor Status Longword indicates the access mode of the currently executing software.

**cylinder** The tracks at the same radius on all recording surfaces of a disk.

**data structure** Any table, list, array, queue, or tree whose format and access conventions are well-defined for reference by one or more images.

**data type** In general, the way in which bits are grouped and interpreted. In reference to the processor instructions, the data type of an operand identifies the size of the operand and the significance of the bits in the operand. Operand data types include: byte, word, longword, and quadword integer, floating and double floating, character string, packed decimal string, and variable-length bit field.

**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 level 16 through 23. Device interrupts can be requested only by devices, controllers, and memories.

**device name** The field in a file specification that identifies the device unit on which a file is stored. Device names also include the mnemonics that identify an I/O peripheral device in a data transfer request. A device name consists of a mnemonic followed by a controller identification letter (if applicable), followed by a unit number (if applicable). 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 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 comparision is needed to locate any data in the cache because any block of main memory data can be placed in only one possible position in the cache. Contrast with fully associative cache.

**displacement deferred indexed mode** An indexed addressing mode in which the base operand specifier uses displacement deferred mode addressing.

**displacement deferred mode** In displacement deferred mode addressing, the specifier extension is a byte, word, or longword displacement. The displacement is sign extended to 32 bits and added to a base address obtained from the specified register. The result is the address of a longword which contains the address of the actual operand. If the PC is used as the register, the updated contents of the PC are used as the base address. The base address is the address of the first byte beyond the specifier extension.

**displacement indexed mode** An indexed addressing mode in which the base operand specifier uses displacement mode addressing.

**displacement mode** In displacement mode addressing, the specifier extension is a byte, word, or longword displacement. The displacement is sign extended to 32 bits and added to a base address obtained from the specified register. The result is the address of the actual operand. If the PC is used as the register, the updated contents of the PC are used as the base address. The base address is the address of the first byte beyond the specifier extension.

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

**drive** The electro-mechanical unit of a mass storage device system on which a recording medium (disk cartridge, disk pack, or magnetic tape reel) is mounted.

**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. An escape character is the code that indicates the transition from normal to escape mode. An escape sequence refers to the set of character combinations starting with an escape character that the terminal transmits without interpretation to the software set up to handle escape sequences.

**event** A change in process status or an indication of the occurrence of some activity that concerns an individual process or cooperating processes. An incident reported to the scheduler that affects a process' ability to execute. Events can be synchronous with the process' execution (a wait request), or they can be asynchronous (I/O completion). Some other events include: swapping; wake request; page fault.

**event flag** A bit in an event flag cluster that can be set or cleared to indicate the occurrence of the event associated with that flag. Event flags are used to synchronize activities in a process or among many processes.

**exception** An event detected by the hardware (other than an interrupt or jump, branch, case, or call instruction) that changes the normal flow of instruction execution. An exception is always caused by the execution of an instruction or set of instructions (whereas an interrupt is caused by an activity in the system independent of the current instruction). There are three types of hardware exceptions: traps, faults, and aborts. Examples are: attempts to execute a privileged or reserved instruction, trace traps, compatibility mode faults, breakpoint instruction execution, and arithmetic traps such as overflow, underflow, and divide by zero.

**exception condition** A hardware- or software-detected event other than an interrupt or jump, branch, case, or call instruction that changes the normal flow of instruction execution.

**exception enables** See trap enables.

**exception vector** See vector.

**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.

**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** Four contiguous bytes (32 bits) starting on an addressable byte boundary. The bits are labeled from right to left from 0 to 31. A two-word floating point number is identified by the address of the byte

containing bit 0. Bit 15 contains the sign of the number. Bits 14 through 7 contain the excess 128 binary exponent. Bits 31 through 16 and 6 through 0 contain a normalized 24-bit fraction with the redundant most significant fraction bit not represented. Within the fraction, bits of decreasing significance go from bit 6 through 0, then 31 through 16. Exponent values of 1 through 255 in the 8-bit exponent field represent true binary exponents of −128 to 127. An exponent value of 0 together with a sign bit of 0 represent a floating value of 0. An exponent value of 0 with a sign bit of 1 is a reserved representation; floating point instructions processing this value return a reserved operand fault. The value of a floating datum is in the approximate range (+ or −) $0.29 \times 10^{-38}$ to $1.7 \times 10^{38}$. The precision is approximately one part in $2^{23}$ or seven decimal digits.

**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.

**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 zeros.

**hardware context** The values contained in the following registers while a process is executing: the Program Counter (PC); the Processor Status Longword (PSL); the 14 general registers (R0 through R13); the four processor registers (P0BR, P0LR, P1BR and P1LR) that describe the process virtual address space; the Stack Pointer (SP) for the current access mode in which the processor is executing; plus the contents to be loaded in the Stack Pointer for every access mode other than the current access mode. While a process is executing, its hardware context is continually being updated by the processor. While a process is not executing its hardware context is stored in its hardware PCB.

**hardware process control block (PCB)** A data structure known to the processor that contains the hardware context when a process is not executing. A process' hardware PCB resides in its process header.

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

**indexed addressing mode** In indexed mode addressing, two registers are used to determine the actual instruction operand: an index register and a base operand specifier. The contents of the index register are used as an index (offset) into a table or array. The base operand specifier supplies the base address of the array (the base operand address or BOA). The address of the actual operand is calculated by multiplying the contents of the index register by the size (in bytes) of the actual operand and adding the result to the base operand address. The addressing modes resulting from index mode addressing are formed by adding the suffix "indexed" to the addressing mode of the base operand specifier: register deferred indexed, autoincrement indexed, autoincrement deferred indexed (or absolute indexed), autodecrement indexed, displacement indexed, and displacement deferred indexed.

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

**input stream** The source of commands and data. One of: 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 pre-fetch instructions in the instruction stream. The control logic continously fetches data from memory to keep the 8-byte buffer full.

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

**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 1 is lowest, 31 highest. The levels arbitrate contention for processor service. For example, a device cannot interrupt the processor if the processor is currently executing at an interrupt priority level greater than the interrupt priority level of the device's interrupt service routine.

**interrupt service routine** The routine executed when a device interrupt occurs.

**interrupt stack** The system-wide stack used when executing in interrupt service context. At any time, the processor is either in a process context executing in user, supervisor, executive or kernel mode, or in system-wide interrupt service context operating with kernel privileges, as indicated by the interrupt stack and current mode bits in the PSL. The interrupt stack is not context switched.

**interrupt stack pointer** The stack pointer for the interrupt stack. Unlike the stack pointers for process context stacks, which are stored in the hardware PCB, the interrupt stack pointer is stored in an internal register.

**interrupt vector** See vector.

**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, or quadword, the operand is zero extended and can express values in the range 0 through 63 (decimal). If the operand data type is floating or double floating, the 6-bit field is composed of two 3-bit fields, one for the exponent and the other for the fraction. The operand is extended to floating or double floating format.

**longword** Four contiguous bytes (32 bits) starting on an addressable byte boundary. Bits are numbered from right to left with 0 through 31. The address of the longword is the address of the byte containing bit 0. When interpreted arithmetically, a longword is a two's complement integer with significance increasing from bit 0 to bit 30. When interpreted as a signed integer, bit 31 is the sign bit. The value of the signed integer is in the range -2,147,483,648 to

2,147,483,647. When interpreted as an unsigned integer, significance increases from bit 0 to bit 31. The value of the unsigned integer is in the range 0 through 4,294,967,295.

**main memory** See physical memory.

**mass storage device** A device capable of reading and writing data on mass storage media such as a disk pack or a magnetic tape reel.

**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** The processor's primary execution mode in which the programmed instructions are interpreted as byte-aligned, variable-length instructions that operate on byte, word, longword, and quadword integer, floating and double floating, character string, packed decimal, and variable-length bit field data. The instruction execution mode other than compatibility mode.

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

**numeric string** A contiguous sequence of bytes representing up to 31 decimal digits (one per byte) and possibly a sign. The numeric string is specified by its lowest addressed location, its length, and its sign representation.

**offset** A fixed displacement from the beginning of a data structure. System offsets for items within a data structure normally have an associated symbolic name used instead of the numeric displacement. Where symbols are defined, programmers always reference the symbolic names for items in a data structure instead of using the numeric displacement.

**opcode** The pattern of bits within an instruction that specify the operation to be performed.

**operand specifier** The pattern of bits in an instruction that indicate the addressing mode, a register and/or displacement, which, taken together, identify an instruction operand.

**operand specifier type** The access type and data type of an instruction's operand(s). For example, the test instructions are of read access type, since they only read the value of the operand. The operand can be of byte, word, or longword data type, depending on whether the opcode is for the TSTB (test byte), TSTW (test word), or TSTL (test longword) instruction.

**packed decimal** A method of representing a decimal number by storing a pair of decimal digits in one byte, taking advantage of the fact that only four bits are required to represent the numbers zero through nine.

**packed decimal string** A contiguous sequence of up to 16 bytes interpreted as a string of nibbles. Each nibble represents a digit except the low-order nibble of the highest addressed byte, which represents the sign. The packed decimal string is specified by its lowest addressed location and the number of digits.

**page** 1. A set of 512 contiguous byte locations used as the unit of memory mapping and protection. 2. The data between the beginning of file and a page marker, between two markers, or between a marker and the end of a file.

**page 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 frame number needed to map the virtual page to a physical page. When it is not in memory, the page table entry contains the information needed to locate the page on secondary storage (disk).

**paging** The action of bringing pages of an executing process into physical memory when referenced. When a process executes, all of its pages are said to reside in virtual memory. Only the actively used pages, however, need to reside in physical memory. The remaining pages can reside on disk until they are needed in physical memory. In this system, a process is paged only when it references more pages than it is allowed to have in its working set. When the process refers to a page not in its working set, a page fault occurs. This causes the operating system's pager to read in the referenced page if it is on disk (and, optionally, other related pages depending on a cluster factor), replacing the least recently faulted pages as needed. A process pages only against itself.

**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 possitle 3-bit physical addesses 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 SBI that are used to store: 1) instructions that the processor can directly fetch and execute, and 2) any other data that a processor is instructed to manipulate. Also called main memory.

**position dependent code** Code that can execute properly only in the locations in virtual address space that are assigned to it by the linker.

**position independent code** Code that can execute properly without modification wherever it is located in virtual address space, even if its location is changed after it has been linked. Generally, this code uses addresssing modes that form an effective address relative to the PC.

**privileged instructions** In general, any instructions intended for use by the operating system or privileged system programs. In particular, instructions that the processor will not execute unless the current access mode is kernel mode (e.g., HALT, SVPCTX, LDPCTX, MTPR, and MFPR).

**procedure** 1. A routine entered via a call instruction. 2. See command procedure.

**process** The basic entity scheduled by the system software that provides the context in which an image executes. A process consists of an address space and both hardware and software context.

**process address space** See process space.

**process context** The hardware and software contexts of a process.

**process control block (PCB)** A data structure used to contain process context. The hardware PCB contains the hardware context. The software PCB contains the software context, which includes a pointer to the hardware PCB.

**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.

**process page tables** The page tables used to describe process virtual memory.

**process space** The lowest-addressed half of virtual address space, where per-process instructions and data reside. Process space is divided into a program region and a control region.

**Program Counter (PC)** General register 15 (R15). At the beginning of an instruction's execution, the PC normally contains the address of a location in memory from which the processor will fetch the next instruction it will execute.

**program locality** A characteristic of a program that indicates how close or far apart the references to locations in virtual memory are over time. A program with a high degree of locality does not refer to many widely scattered virtual addresses in a short period of time.

**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** 1. n. A circular, doubly-linked list. See system queues. v. To make an entry in a list or table, perhaps using the INSQUE instruction. 2. See job queue.

**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 level 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 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)** The part of the hardware that interconnects the processor, memory controllers, MASSBUS adaptors, the UNIBUS adaptor.

**system** In the context "system, owner, group, world," the system refers to the group numbers that are used by operating system and its controlling users, the system operators and system manager.

**system address space** See system space and system region.

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

**System Control Block (SCB)** The data structure in system space that contains all the interrupt and exception vectors known to the system.

**System Control Block Base register (SCBB)** A processor register contain-

ing 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 SBR.

**system region** The third quarter of virtual address space. The lowest-addressed half of system space. Virtual addresses in the system region are sharable between processes. Some of the data structures mapped by system region virtual addresses are: system entry vectors, the system control block (SCB), the system page table (SPT), and process page tables.

**system space** The highest-addressed half of virtual address space. See also system region.

**system virtual address** A virtual address identifying a location mapped by an address in system space.

**system virtual space** See system space.

**terminal** The general name for those peripheral devices that have keyboards and video screens or printers. Under program control, a terminal enables people to type commands and data on the keyboard and receive messages on the video screen or printer. Examples of terminals are the LA36 DECwriter hard-copy terminal and VT52 video display terminal.

**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 conditon with a single instruction.

**trap enables** Three bits in the Processor Status Word that control the processor's action on certain arithmetic exceptions.

**two's complement** A binary representation for integers in which a negative number is one greater than the bit complement of the positive number.

**two-way associative cache** A cache organization which has two groups of directly mapped blocks. Each group contains several blocks for each index position in the cache. A block of data from main memory can go into any group at its proper index position. A two-way associative cache is a compromise between the extremes of fully associative and direct mapping cache organizations that takes advantage of the features of both.

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

**unwind the call stack** To remove call frames from the stack by tracing back through nested procedure calls using the current contents of the FP register and the FP register contents stored on the stack for each call frame.

**urgent interrupt** An interrupt received on interrupt priority levels 24 through 31. These can be generated only by the processor for the interval clock, serious errors, and power fail.

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

**user privileges** The privileges granted a user by the system manager. See process privileges.

**value return registers** The general registers R0 and R1 used by convention to return function values. These registers are not preserved by any called procedures. They are available as temporary registers to any called procedure. All other registers (R2, R3,..., R11, AP, FP, SP, PC) are preserved across procedure calls.

**variable-length bit field** A set of zero to 32 contiguous bits located arbitrarily with respect to byte boundaries. A variable bit field is specified by four attributes: 1) the address A of a byte, 2) the bit position P of the starting location of the bit field with respect to bit 0 of the byte at address A, 3) the size, in bits, of the bit field, and 4) whether the field is signed or unsigned.

**vector** 1. A interrupt or exception vector is a storage location known to the system that contains the starting address of a procedure to be executed when a given interrupt or exception occurs. The system defines separate vectors for each interrupting device controller and for classes of exceptions. Each system vector is a longword. 2. For the purposes of exception handling, users can declare up to two software exception vectors (primary and secondary) for each of the four access modes. Each vector contains the address of a condition handler. 3. A one-dimensional array.

**virtual address** A 32-bit integer identifying a byte "location" in virtual address space. The memory management hardware translates a virtual address to a physical address. The term virtual address may also refer to the address used to identify a virtual block on a mass storage device.

**virtual address space** The set of all possible virtual addresses that an image executing in the context of a process can use to identify the location of an instruction or data. The virtual address space seen by the programmer is a linear array of 4,294,967,296 ($2^{32}$) byte addresses.

**virtual memory** The set of storage locations in physical memory and on disk that are referred to by virtual addresses. From the programmer's viewpoint, the secondary storage locations appear to be locations in physical memory. The size of virtual memory in any system depends on the amount of physical memory available and the amount of disk storage used for non-resident virtual memory.

**virtual page number** The virtual address of a page of virtual memory.

**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.

**write access type** The specified operand of an instruction or procedure is only written during that instruction's or procedure's execution.

**write allocate** A cache management technique in which cache is allocated on a write miss as well as on the usual read miss.

**write back** A cache management technique in which data from a write operation to cache 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

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 manual? (format, accuracy, completeness, organization, etc.) _____

What features are most useful? _____

Does the publication satisfy your needs? _____

What errors have you found? _____

Additional comments _____

Name _____

Company Dept. _____

Title _____

City State Zip _____

(staple here)

— — — — — — — (please fold here) — — — — — — — —

FIRST CLASS
PERMIT NO. 33
MAYNARD, MASS.

**BUSINESS REPLY MAIL**
**NO POSTAGE STAMP NECESSARY**
**IF MAILED IN THE UNITED STATES**

Postage will be paid by:

**DIGITAL EQUIPMENT CORPORATION**
**PRODUCT PROMOTION GROUP**
**PK3-2/M18**
**MAYNARD, MASS. 01754**

(staple here)

digital

DIGITAL EQUIPMENT CORPORATION, Corporate Headquarters: Maynard, Massachusetts 01754, Telephone: (617)897-5111—SALES AND SERVICE OFFICES: UNITED STATES—ALABAMA, Huntsville • ARIZONA, Phoenix and Tucson • CALIFORNIA, El Segundo, Los Angeles, Oakland, Ridgecrest, San Diego, San Francisco (Mountain View), Santa Ana, Santa Clara, Stanford, Sunnyvale and Woodland Hills • COLORADO, Englewood • CONNECTICUT, Fairfield and Meriden • DISTRICT OF COLUMBIA, Washington (Lanham, MD) • FLORIDA, Ft. Lauderdale and Orlando • GEORGIA, Atlanta • HAWAII, Honolulu • ILLINOIS, Chicago (Rolling Meadows) • INDIANA, Indianapolis • IOWA, Bettendorf • KENTUCKY, Louisville • LOUISIANA, New Orleans (Metairie) • MARYLAND, Odenton • MASSACHUSETTS, Marlborough, Waltham and Westfield • MICHIGAN, Detroit (Farmington Hills) • MINNESOTA, Minneapolis • MISSOURI, Kansas City (Independence) and St. Louis • NEW HAMPSHIRE, Manchester • NEW JERSEY, Cherry Hill, Fairfield, Metuchen and Princeton • NEW MEXICO, Albuquerque • NEW YORK, Albany, Buffalo (Cheektowaga), Long Island (Huntington Station), Manhattan, Rochester and Syracuse • NORTH CAROLINA, Durham/Chapel Hill • OHIO, Cleveland (Euclid), Columbus and Dayton • OKLAHOMA, Tulsa • OREGON, Eugene and Portland • PENNSYLVANIA, Allentown, Philadelphia (Bluebell) and Pittsburgh • SOUTH CAROLINA, Columbia • TENNESSEE, Knoxville and Nashville • TEXAS, Austin, Dallas and Houston • UTAH, Salt Lake City • VIRGINIA, Richmond • WASHINGTON, Bellevue • WISCONSIN, Milwaukee (Brookfield) • INTERNATIONAL —ARGENTINA, Buenos Aires • AUSTRALIA, Adelaide, Brisbane, Canberra, Melbourne, Perth and Sydney • AUSTRIA, Vienna • BELGIUM, Brussels • BOLIVIA, La Paz • BRAZIL, Rio de Janeiro and Sao Paulo • CANADA, Calgary, Edmonton, Halifax, London, Montreal, Ottawa, Toronto, Vancouver and Winnipeg • CHILE, Santiago • DENMARK, Copenhagen • FINLAND, Helsinki • FRANCE, Lyon, Grenoble and Paris • GERMAN FEDERAL REPUBLIC, Cologne, Frankfurt, Hamburg, Hannover, Munich, Nuremburg, Stuttgart and West Berlin • HONG KONG • INDIA, Bombay • INDONESIA, Djakarta • IRELAND, Dublin • ITALY, Milan, Rome and Turin • IRAN, Tehran • JAPAN, Osaka and Tokyo • MALAYSIA, Kuala Lumpur • MEXICO, Mexico City • NETHERLANDS, Utrecht • NEW ZEALAND, Auckland and Christchurch • NORWAY, Oslo • PUERTO RICO, Santurce • SINGAPORE • SPAIN, Madrid • SWEDEN, Gothenburg and Stockholm • SWITZERLAND, Geneva and Zurich • UNITED KINGDOM, Birmingham, Bristol, Epsom, Edinburgh, Leeds, Leicester, London, Manchester and Reading • VENEZUELA, Caracas •

PRINTED IN USA EB 07466 20/77 110 23 031