CORPORATE PROFILE

Digital Equipment Corporation designs, manufactures, sells and services computers and associated peripheral equipment, and related software and supplies. The Company's products are used world-wide in a wide variety of applications and programs, including scientific research, computation, communications, education, data analysis, industrial control, timesharing, commercial data processing, word processing, health care, instrumentation, engineering and simulation.
VAX11

ARCHITECTURE HANDBOOK

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

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

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

Copyright © 1979, by Digital Equipment Corporation
CONTENTS

CHAPTER 1  VAX-11 FAMILY ........................................... 1
INTRODUCTION .................................................. 1
THE VAX FAMILY CONCEPT/VAX ARCHITECTURE ............... 1
ARCHITECTURAL GOALS ...................................... 2
NOTATIONAL CONVENTIONS .................................. 3

CHAPTER 2  VAX-11 ARCHITECTURE OVERVIEW ................. 7
INTRODUCTION .................................................. 7
PROCESS VIRTUAL ADDRESS SPACE ............................. 8
DATA TYPES .................................................... 8
GENERAL REGISTERS AND ADDRESSING MODES .............. 10
STACKS, SUBROUTINES, AND PROCEDURES .................. 12
STACK POINTER, ARGUMENT POINTER, AND FRAME POINTER .. 13
PROCESSOR STATUS WORD ................................... 14
INSTRUCTION FORMAT ....................................... 15
INSTRUCTION SET ............................................ 16
NATIVE INSTRUCTION SET .................................... 16
SEPARATION OF PROCEDURE AND DATA ....................... 24
EXCEPTIONS ..................................................... 25
COMPATIBILITY MODE ......................................... 25
PROCESSING CONCEPTS FOR SYSTEM PROGRAMMING .......... 26
SYSTEM PROGRAMMING ENVIRONMENT ....................... 28
FOR MORE INFORMATION ON VAX-11 ARCHITECTURE .......... 39

CHAPTER 3  MEMORY, REGISTERS, AND PROCESSOR STATUS .... 43
INTRODUCTION .................................................. 43
MEMORY ....................................................... 43
GENERAL REGISTERS ......................................... 47
STACKS .......................................................... 50
PROCESSOR STATUS LONGWORD ............................. 52

CHAPTER 4  DATA REPRESENTATION ............................... 57
INTRODUCTION .................................................. 57
<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>INTEGER AND FLOATING DATA TYPES</td>
<td>57</td>
</tr>
<tr>
<td>CHARACTER STRING DATA TYPE</td>
<td>62</td>
</tr>
<tr>
<td>NUMERIC STRING DATA TYPE</td>
<td>63</td>
</tr>
<tr>
<td>PACKED DECIMAL STRING DATA TYPE</td>
<td>68</td>
</tr>
<tr>
<td>QUEUE DATA TYPES</td>
<td>70</td>
</tr>
<tr>
<td>VARIABLE LENGTH BIT FIELD DATA TYPES</td>
<td>72</td>
</tr>
<tr>
<td>DATA IN REGISTERS</td>
<td>75</td>
</tr>
</tbody>
</table>

### CHAPTER 5 INSTRUCTION FORMATS AND ADDRESSING MODES

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>INTRODUCTION</td>
<td>79</td>
</tr>
<tr>
<td>GENERAL REGISTERS</td>
<td>79</td>
</tr>
<tr>
<td>INSTRUCTION FORMAT</td>
<td>80</td>
</tr>
<tr>
<td>ADDRESSING MODE</td>
<td>83</td>
</tr>
<tr>
<td>GENERAL MODE ADDRESSING</td>
<td>85</td>
</tr>
<tr>
<td>Register Mode</td>
<td>85</td>
</tr>
<tr>
<td>Register Deferred Mode</td>
<td>86</td>
</tr>
<tr>
<td>Autoincrement Mode</td>
<td>88</td>
</tr>
<tr>
<td>Autoincrement Deferred Mode</td>
<td>90</td>
</tr>
<tr>
<td>Autodecrement Mode</td>
<td>92</td>
</tr>
<tr>
<td>Literal Mode</td>
<td>93</td>
</tr>
<tr>
<td>Displacement Mode</td>
<td>97</td>
</tr>
<tr>
<td>Displacement Deferred Mode</td>
<td>99</td>
</tr>
<tr>
<td>INDEX MODE</td>
<td>100</td>
</tr>
<tr>
<td>PROGRAM COUNTER ADDRESSING MODE</td>
<td>110</td>
</tr>
<tr>
<td>Immediate Mode</td>
<td>111</td>
</tr>
<tr>
<td>Absolute Mode</td>
<td>112</td>
</tr>
<tr>
<td>Relative Mode</td>
<td>114</td>
</tr>
<tr>
<td>Relative Deferred Mode</td>
<td>115</td>
</tr>
<tr>
<td>BRANCH ADDRESSING MODE</td>
<td>117</td>
</tr>
</tbody>
</table>

### CHAPTER 6 INTEGER AND FLOATING POINT INSTRUCTIONS

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>INSTRUCTION SET OVERVIEW</td>
<td>121</td>
</tr>
<tr>
<td>FLOATING POINT INSTRUCTIONS</td>
<td>121</td>
</tr>
<tr>
<td>MOV, PUSHL, CLR, MNEG, MCOM, CVT, MOVZ,</td>
<td>124</td>
</tr>
<tr>
<td>CMP, INC, TST, ADD, ADWC, ADAWI, SUB, DEC,</td>
<td></td>
</tr>
<tr>
<td>SBWC, MUL, EMUL, EMOD, DIV, EDIV, BIT,</td>
<td></td>
</tr>
<tr>
<td>BIS, BIC, XOR, ASH, ROTL, POLY</td>
<td></td>
</tr>
</tbody>
</table>
### CHAPTER 7 SPECIAL INSTRUCTIONS
- INTRODUCTION ................................................. 165
- MULTIPLE REGISTER INSTRUCTIONS ...................... 165
  - PUSHR, POPR  .............................................. 165
  - PROCESSOR STATUS LONGWORD MANIPULATION ............. 169
  - MOVPSL, BISPSW, BICPSW  .................................. 169
- ADDRESS INSTRUCTIONS ...................................... 171
  - MOVA, PUSH ................................................. 171
- INDEX INSTRUCTION ........................................... 173
  - INDEX ....................................................... 173
- QUEUE INSTRUCTIONS ......................................... 176
  - Absolute Queues ......................................... 176
  - INSQUE, REMQUE ........................................... 176
- Self-Relative Queues ......................................... 184
  - INSQHI, INSQTI, REMQHI, REMQTI .......................... 184
- VARIABLE LENGTH BIT FIELD INSTRUCTIONS ................. 196
  - FF, EXT, CMP, INSV ......................................... 196

### CHAPTER 8 CONTROL INSTRUCTIONS
- INTRODUCTION ................................................. 205
- BRANCH AND JUMP INSTRUCTIONS ............................. 205
  - B, BR, JMP, BB, BLB ....................................... 205
- LOOP CONTROL INSTRUCTIONS ................................. 213
  - ACB, AOB, SOB ............................................. 213
- CASE INSTRUCTION ............................................ 218
  - CASE ....................................................... 218
- SUBROUTINE INSTRUCTIONS ................................... 220
  - BSB, JSB, RSB .............................................. 220
- PROCEDURE CALL INSTRUCTIONS ............................... 222
  - CALLG, CALLS, RET ....................................... 222

### CHAPTER 9 CHARACTER STRING INSTRUCTIONS
- INTRODUCTION ................................................. 233
- CHARACTER STRING INSTRUCTIONS ......................... 233
  - MOVC, MOVTC, MOVTUC, CMPC, SCANC, SPANC, LOCC, SKPC, MATCHC  
  - CYCLIC REDUNDANCY CHECK INSTRUCTION ................... 245
  - CRC ....................................................... 245

### CHAPTER 10 DECIMAL STRING INSTRUCTIONS
- INTRODUCTION ................................................. 251
| DECIMAL OVERFLOW                           | 252 |
| ZERO NUMBERS                              | 252 |
| RESERVED OPERAND EXCEPTION                | 252 |
| UNPREDICTABLE RESULTS                     | 252 |
| PACKED DECIMAL OPERATIONS                 | 252 |
| ZERO LENGTH DECIMAL STRINGS               | 253 |
| MOVDP, CMPP, ADDP, SUBP, MULP, DIVP        |     |
| CVTLP, CVTPL, CVTP, CVTTP, CVTPS, CVTSP, CVTSP, ASHP |     |

### CHAPTER 11 EDIT INSTRUCTION

| INTRODUCTION                               | 275 |
| EDITPC, EO$INSERT, EO$STORE_SIGN,         |     |
| EO$FILL, EO$MOVE, EO$FLOAT, EO$END_FLOAT  |     |
| EO$BLANK_ZERO, EO$REPLACE_SIGN, EO$LOAD   |     |
| EO$_SIGNIF, EO$ADJUST_INPUT, EO$END        |     |

### CHAPTER 12 EXCEPTIONS

| INTRODUCTION                               | 295 |
| PROCESSOR STATUS                          | 296 |
| ARITHMETIC TRAPS                          | 299 |
| EXCEPTIONS DETECTED DURING OPERAND REFERENCE | 302 |
| EXCEPTIONS OCCURRING AS THE               |     |
| CONSEQUENCE OF AN INSTRUCTION START       | 304 |
| TRACING                                   | 305 |
| SERIOUS SYSTEM FAILURES                   | 308 |
| STACKS                                    | 309 |
| RELATED INSTRUCTIONS                      | 311 |
| REI, BPT, HALT                            |     |

### CHAPTER 13 PDP-11 COMPATIBILITY MODE

| INTRODUCTION                               | 317 |
| COMPATIBILITY MODE USER ENVIRONMENT        | 317 |
| ENTERING AND LEAVING COMPATIBILITY MODE     | 320 |
| COMPATIBILITY MODE MEMORY MANAGEMENT       | 321 |
| COMPATIBILITY MODE EXCEPTIONS AND INTERRUPTS| 323 |
| T BIT OPERATION IN COMPATIBILITY MODE       | 324 |
| UNIMPLEMENTED PDP-11 TRAPS                  | 326 |
| COMPATIBILITY MODE I/O REFERENCES           | 326 |
| PROCESSOR REGISTERS                        | 326 |
| PROGRAM SYNCHRONIZATION                    | 326 |
VAX-11 is DIGITAL's 32-bit extension to its 11 family of minicomputers. VAX-11 is a fully integrated computer system featuring state-of-the-art hardware technology coupled with a powerful virtual memory operating system, VAX/VMS (Virtual Address Extension/Virtual Memory System). VAX-11 hardware is characterized by its flexible instruction set, 32-bit capability, byte addressability, stack orientation, and highly efficient page-oriented memory management scheme. VAX/VMS is a high-performance operating system designed to complement the VAX-11 hardware. VAX/VMS encompasses a highly sensitive scheduling algorithm, extensive record and file management capabilities, and virtual memory features achieved by an extremely efficient paging algorithm.

VAX-11 is general purpose in nature, with the inherent capability to deal with a multitude of user environments. Designed to optimize throughput, the system enables enormous amounts of data to flow through it swiftly and unobstructedly. VAX-11 supports a 32-bit word architecture, thereby establishing a virtual address space of \(2^{32}\) or 4.3 billion bytes for user application. The native mode instruction set provides over 240 instructions and is highly bit-efficient. The instruction set includes integral decimal, character string, and floating point instructions, as well as integer, logical and bit field instructions. Instructions and data are variable length and can start at any arbitrary byte boundary. In the case of bit fields, they can start at any arbitrary bit in memory.

The VAX-11 provides sixteen 32-bit general registers that can be used for temporary storage, as accumulators, as index registers, and as base registers. Four registers have special significance: the Program Counter, and three registers that are used to provide an extensive CALL facility. The processor offers a variety of addressing modes that use the general registers to identify instruction operand locations, including an indexed addressing mode that provides a true post-indexing capability.

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

The VAX-11 handbook documentation set is presented in three books:

- The VAX-11 Software Handbook introduces the VAX/VMS virtual memory operating system, its operation, hardware interaction, associated software, data structures, features, and capabilities.
• The VAX-11 Hardware Handbook introduces the VAX-11 hardware elements, including the I/O subsystem, the central processor unit, the console subsystem, MASSBUS and UNIBUS systems, main memory, and memory management.

• This book, the VAX-11 Architecture Handbook, introduces VAX-11 architecture, addressing modes, and the native mode instruction set.
CHAPTER 1
VAX-11 FAMILY

INTRODUCTION
The VAX-11 family represents a significant extension of the PDP-11 family architecture. The term VAX is taken from the single most significant feature of the VAX-11 family, Virtual Address eXtension. It shares with the PDP-11 byte addressing, similar I/O and interrupt structures, and identical data formats. In addition to a greatly extended virtual address space, VAX-11 offers additional instructions and data types, and new addressing modes. Also provided is a sophisticated memory management and protection mechanism, and hardware assisted process scheduling and synchronization.

Although the VAX native mode instruction set is not compatible with the PDP-11, it is a logical extension to the PDP-11 instruction set, sharing the PDP's ease of programming. The similarity between VAX-11 and PDP-11 enables straightforward manual conversion of existing PDP-11 programs to VAX-11. Most existing user mode PDP-11 programs which do not need the extended features of VAX-11 can run unchanged in the PDP-11 compatibility mode provided in VAX-11.

VAX-11 is designed for applications that require the power and sophistication of a high-performance, virtual memory computer system. 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.

THE VAX FAMILY CONCEPT/VAX ARCHITECTURE
An important objective from the outset has been to assure that all VAX programs execute on each member of the VAX-11 family. The collection of attributes that all family members have in common which assures this software compatibility is collectively referred to as the VAX architecture. As new computers are added to the VAX family, a significant part of the engineering effort will be dedicated to preserving compatibility. This process, referred to as Architecture Control, will assure that programs written for today's VAX-11 computers will execute on future VAX-11s. This handbook documents those aspects which programmers should understand to write applications pro-
grams for any VAX-11 computer. It is important to emphasize that the VAX-11 architecture is the definition of an environment, as opposed to the implementation of it. For example, the architecture provided to the software includes the instruction set, the addressing modes, data types, etc. Attributes not included in the architecture include the internal bus structure, implementation-specific privileged registers, execution speed, etc. These implementation-specific details are documented for each processor in the appropriate hardware handbook.

**ARCHITECTURAL GOALS**

The VAX-11 architecture was designed around a number of specific user-oriented goals. These were:

- Extension of the virtual address space, (This was the single most important goal.) The design decision was between a 24-bit virtual address (capable of addressing 16 million bytes) and a 32-bit virtual address (capable of addressing 4 billion bytes). Since the main reason for going to new architectures in the past, as with VAX-11, has been the need for larger virtual addresses, it was felt that 24 bits would be extremely short-sighted. It was decided that a 32-bit virtual address size would be required in order for the architecture to span the 1980's.

- Maximize compatibility with the PDP-11 consistent with a significant extension of the virtual address space, and a significant functional enhancement. The goal of compatibility with the PDP-11 was met in several ways:

  1. The formats for representing data are the same.
  2. The media formats for peripheral I/O devices are the same. Combined with the same data formats, this means that all data files can be read by both PDP-11 and VAX-11 systems.
  3. The assembly language syntax and mnemonics are basically the same. It has been very easy for PDP-11 trained programmers to learn to write VAX-11 code.
  4. In addition to making it easy to convert PDP-11 programs to run on the VAX-11, the VAX-11 architecture includes a PDP-11 compatibility mode in hardware, which allows user-mode PDP-11 programs to run unchanged. This allows, for example, a user's old application written for the PDP-11 to run simultaneously with VAX native mode programs.

- Promote high bit efficiency by offering a wide range of data types and addressing modes.

- Architectural extensibility—the instruction set is designed so that new data types and operators can be efficiently included in a manner consistent with the currently defined operators and data types.
The instruction format that was selected allows for effectively unlimited flexibility as it becomes desirable to add features in the future. Thus, the current definition of the architecture will not significantly constrain future enhancements.

- The goal of easy use by higher level languages was broken down into several specific requirements:

1. Orthogonality of operation, data type and addressing mode. This means that the operation being performed, (e.g. ADD, SUB, etc.), the type of data (integer, floating point, etc.) and method of addressing (indirect, indexed, etc.) can be considered independently by the compiler, which makes compilers faster, more efficient, and easier to implement.

2. No forced alignment on longword boundary. This means that data items larger than a byte can still be on any byte boundary, as is required in some languages (such as the FORTRAN COMMON facility).

3. Variable number of operands. By picking an instruction format which allows each instruction to have its “natural” number of operands, it is possible to have instructions in the format desirable for compilers. E.g., A + B → C is ADDL3, A, B, C while A + B → B is ADDL2 A, B.

4. Appropriate high level primitives. The operations commonly found in higher level languages are often built directly into hardware. For example, 3-address arithmetic (A = B + C), loop control (the FORTRAN DO and BASIC FOR loops are one instruction), and output formatting (the EDIT instruction can be used for COBOL PICTURE statements).

- In addition, instructions were added to make various applications and operating systems code more efficient. Examples of these are hardware support of queues, easy access to variable length bit fields, and simple instructions to save and restore program context.

- A final major goal was to make the architecture appropriate over wide ranges of system cost, performance, and applications. This would allow a wide range of user needs to be served by a single architecture and eliminate unnecessary spending associated with the support of many different architectures.

**NOTATIONAL CONVENTIONS**

This section provides information on notational conventions used throughout the handbook set.
**Operational Notation**

Representations of memory, both physical and virtual, begin with low memory at the top of the diagram and progress toward higher addresses. This concept is illustrated in Figure 1-1.

![Figure 1-1 Memory Addressing Scheme](image)

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:

\[ \text{sum} \leftarrow \text{sum} + \text{add} + C \]

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.

**Ranges and Extents**

Ranges are specified in English by the word “through”, in notational form by a double period “..” and are inclusive. For example, the range 0 through 4 or 0..4 includes the integers 0,1,2,3 and 4.

Extents are specified by a pair of numbers separated by a colon and are inclusive. For example, bits 7:3 specify an extent of bits including bits 7,6,5,4 and 3.

**Unpredictable and Undefined**

Results specified as unpredictable may vary from moment to moment, implementation to implementation, and instruction to instruction within an implementation. Engineering change orders (ECO’s) may change unpredictable results. Software should never depend on results specified as unpredictable.

Operations specified as undefined may vary from moment to moment, implementation to implementation, and instruction to instruction with-
in an implementation. Engineering change orders may change unde-
defined operations. The operation may vary in effect from nothing to
stopping system operation. Non-privileged software should avoid in-
voking undefined operations.

**MBZ and Reserved**

Fields specified as MBZ (Must Be Zero) should never be filled by
software with a non-zero value. If the processor encounters a non-zero
value in a field specified as MBZ, generally a reserved operand fault or
abort occurs.

Certain fields and values accessible to privileged software are
reserved to DIGITAL and the privileged software should not set non-
zero or reserved values into these areas.

In some cases, certain unassigned values are indicated as “reserved
to CSS and customers”. Only these values should be used for non-
standard applications. The values indicated as reserved to DIGITAL
and all MBZ fields are to be used only to extend the standard architec-
ture in the future.
CHAPTER 2
VAX-11 ARCHITECTURE OVERVIEW

INTRODUCTION
This chapter presents an overview of the VAX-11 Family Architecture. The term “VAX-11 Family Architecture” can be abbreviated to simply “VAX Architecture”. In either case, what is meant are the attributes of a system as seen by the assembly language programmer. I.e., the conceptual structure and functional behavior, as distinct from the organization of the data flow and controls, the logical design, and the physical implementation. Thus, the VAX-11 family architecture defines the consistent functional behavior seen by a programmer on all processors of the family. The primary advantage of a common family architecture is the ability to execute software on any VAX family member. The VAX-11 family architecture is divided into two distinct areas: application programmer aspects and system programmer aspects. Application programmer architecture is described in detail in this handbook while details of the system programmer architecture, beyond this overview, can currently be found in the appropriate hardware handbook.

Those attributes of the VAX-11 family that are part of the user architecture are:

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

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

- Privileged instructions
- High word of the Processor Status Longword
- Process structure
- Memory management
- Interrupt structure and exception handling
Those attributes of VAX-11 that are processor-specific (not defined as part of the family, architecturally) are:

- Features of the writable control store (WCS) and micro machine
- Size and availability of cache memory
- Size of translation buffer
- Bus widths, electrical specifications, protocols, bandwidth (throughput), etc.
- Accelerators
- LSI console with floppy (VAX-11/780 specific)
- Main memory limitation
- Certain processor registers used for implementation-specific information required by diagnostics, such as error and status registers.

**PROCESS VIRTUAL ADDRESS SPACE**

Most data are located in memory using the address of an 8-bit byte. The programmer uses a 32-bit virtual address to identify a byte location. This address is called a virtual address because it is not the real address for a physical memory location. It is translated into a real address by the processor under operating system control. A virtual address is not a unique address of a location in memory, as are physical memory addresses. Two programs using the same virtual address might refer to two different physical memory locations, or refer to the same physical memory location using different virtual addresses.

The set of all possible 32-bit virtual addresses is called virtual address space. It can be viewed as an array of byte "locations" labelled from zero to $2^{32} - 1$, an array that is approximately four billion bytes in length. This address space is divided into sets of virtual addresses designated for certain uses. The set of virtual addresses designated for use by a process is one half of the total virtual address space. Addresses in the remaining half of virtual address space are used to refer to locations maintained and protected by the operating system.

**DATA TYPES**

The data type of an instruction operand identifies how many bits of storage are to be treated as a unit, and what the interpretation of that unit is. The processor's native instruction set recognizes six primary data types: integer and floating, character string, packed decimal, numeric string, and variable length bit field. For each of these data types, the selection of operation immediately informs the processor of the size and interpretation of the data. The processor can manipulate the bit field, as a function of user defined field size and relative position from a given byte address.
There are several variations on the six primary data types. Table 2-1 provides a summary of the data types available, some of which are illustrated in Figure 2-1. Integer data are stored as binary values. An integer can be stored as a byte, word, longword, or, in some cases, as a quadword. A byte is eight bits, a word is two bytes, a longword is four bytes, and a quadword is eight bytes. The processor can interpret an integer as either a signed (2's complement) value or an unsigned value. The sign is determined by the high-order bit.

**Table 2-1 Data Types**

<table>
<thead>
<tr>
<th>DATA TYPE</th>
<th>SIZE</th>
<th>RANGE (decimal)</th>
<th>Unsigned</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integer</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Byte</td>
<td>8 bits</td>
<td>-128 to +127</td>
<td>0 to 255</td>
</tr>
<tr>
<td>Word</td>
<td>16 bits</td>
<td>-32768 to +32767</td>
<td>0 to 65535</td>
</tr>
<tr>
<td>Longword</td>
<td>32 bits</td>
<td>-231 to +231-1</td>
<td>0 to 232-1</td>
</tr>
<tr>
<td>Quadword</td>
<td>64 bits</td>
<td>-263 to +263-1</td>
<td>0 to 264-1</td>
</tr>
<tr>
<td>Floating Point</td>
<td>±2.9 × 10^-3 to 1.7 × 10^38</td>
<td></td>
<td></td>
</tr>
<tr>
<td>F_floating</td>
<td>32 bits</td>
<td>approximately seven decimal digits precision</td>
<td></td>
</tr>
<tr>
<td>D_floating</td>
<td>64 bits</td>
<td>approximately sixteen decimal digits precision</td>
<td></td>
</tr>
<tr>
<td>Packed Decimal String</td>
<td>0 to 16 bytes (31 digits)</td>
<td>numeric, two digits per byte sign in low half of last byte</td>
<td></td>
</tr>
<tr>
<td>Character String</td>
<td>0 to 65535 bytes</td>
<td>one character per byte</td>
<td></td>
</tr>
<tr>
<td>Variable-length Bit Field</td>
<td>0 to 32 bits</td>
<td>dependent on interpretation</td>
<td></td>
</tr>
<tr>
<td>Numeric String</td>
<td>0 to 31 bytes (DIGITS)</td>
<td>-1031−1 to +1031−1</td>
<td></td>
</tr>
<tr>
<td>Queue</td>
<td>≥ 2 longwords/queue entry</td>
<td>0 thru 2 billion entries</td>
<td></td>
</tr>
</tbody>
</table>

Floating point values are stored using a signed 8-bit excess-128 exponent and a binary normalized fraction, using 4-byte and 8-byte formats identical to those of the PDP-11. The normalization bit is not represented, to provide an effective 24-bit fraction in the 4-byte format, and an effective 56-bit fraction in the 8-byte format. The 4-byte format, called F_floating, provides approximately seven decimal digits of precision, while the 8-byte format, called D_floating, provides approximately 16 decimal digits of precision.

Packed decimal data are stored in a string of bytes. Each byte is divided into two 4-bit nibbles. One decimal digit is stored in each nibble. The first, or most significant digit is stored in the high-order nibble of the first byte, the second digit is stored in the low-order nibble of the first byte, the third digit is stored in the high-order nibble of the second byte, and so on. The sign of the number is stored in the low-order nibble of the last byte of the string.
Character data are simply a string of bytes containing any binary data, for example, ASCII codes. The first character in the string is stored in the first byte, the second character is stored in the second byte, and so on. A character string that contains ASCII codes for decimal digits is called a numeric string.

The address of any data item is the address of the first byte in which the item resides. All integer, floating point, packed decimal, and character data can be stored starting on an arbitrary byte boundary. A bit field, however, does not necessarily start on a byte boundary. A field is simply a set of contiguous bits (0-32) whose starting bit location is identified relative to a given byte address. The native instruction set can interpret a bit field as a signed or unsigned integer.

**GENERAL REGISTERS AND ADDRESSING MODES**
A general register is a location within the processor that can be used for temporary data storage and addressing. The assembly language programmer has sixteen 32-bit general registers available for use with
the native instruction set. Some of these registers have special significance. For example, one register is designated as the Program Counter, and it contains the address of the next instruction to be executed. Three general registers are designated for use with routine linkages: the Stack Pointer, the Argument Pointer, and the Frame Pointer.

An instruction operand can be located in main memory, in a general register, or in the instruction stream itself. The method in which an operand location is specified is called the operand addressing mode. The processor offers a variety of addressing modes and addressing mode optimizations. There is an addressing mode that locates an operand in a register, and several addressing modes that locate an operand in memory by using a register to:

- point to the operand
- point to a table of operands
- point to a table of operand addresses

There are also addressing modes that are indexed modifications of the addressing modes that locate an operand in memory. Finally, there are addressing modes that identify the location of the operand in the instruction stream including one for constant data and one for branch instruction addresses. The VAX-11 addressing modes are summarized in Table 2-2.

**Table 2-2  Addressing Modes**

<table>
<thead>
<tr>
<th>Literal (Immediate)</th>
<th>{S↑} #constant</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register</td>
<td>Rn</td>
</tr>
<tr>
<td>Register Deferred</td>
<td>(Rn)</td>
</tr>
<tr>
<td>Autodecrement</td>
<td>-(Rn)</td>
</tr>
<tr>
<td>Autoincrement</td>
<td>(Rn) +</td>
</tr>
<tr>
<td>Autoincrement Deferred (Absolute)</td>
<td>@ (Rn) +</td>
</tr>
<tr>
<td></td>
<td>@ # address</td>
</tr>
<tr>
<td>Displacement</td>
<td>{B↑ \ W↑ \ L↑} displacement (Rn)</td>
</tr>
<tr>
<td>Displacement Deferred</td>
<td>@ {B↑ \ W↑ \ L↑} displacement (Rn)</td>
</tr>
</tbody>
</table>

Indexed [Rx]

n = 0 through 15
x = 0 through 14
STACKS, SUBROUTINES, AND PROCEDURES
A stack is an array of consecutively addressed data items that are referenced on a last-in, first-out basis using a general register. Data items are added to and removed from the low address end of the stack. A stack grows toward lower addresses as items are added, and shrinks toward higher addresses as items are removed. Refer to Appendix C (DIGITAL Calling Standard) for a detailed explanation of the VAX Software Architecture.

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

A stack is an extremely powerful data structure because it can be used to pass arguments to routines efficiently. In particular, the stack structure enables the coding of reentrant routines because the processor can handle routine linkages automatically using the Stack Pointer. Routines can also be recursive because arguments can be saved on the stack for each successive call of the same routine.

The processor provides two kinds of routine call instructions: those for subroutines, and those for procedures. In general, a subroutine is a routine entered using a Jump to Subroutine or Branch to Subroutine instruction, while a procedure is a routine entered using a Call instruction.

The processor provides more elaborate routine linkages for procedures than for subroutines. The processor automatically saves and restores the contents of registers to be preserved across procedure calls. The processor provides two methods for passing argument lists to called procedures: by passing the arguments on the stack, or by passing the address of the arguments elsewhere in memory. The processor also constructs a "journal" of procedure call nesting by using a general register as a pointer to the place on the stack where a procedure has its linkage data. This record of each procedure's stack data, known as its stack frame, enables proper returns from procedures even when a procedure leaves data on the stack. In addition, user and operating system software can use the stack frame to trace back through nested calls to handle errors or debug programs.
STACK POINTER, ARGUMENT POINTER AND FRAME POINTER
The Stack Pointer is a register specifically designated for use with stack structures. By use of autodecrement mode addressing through the Stack Pointer, items can be placed on the stack and removed by use of the autoincrement mode. The top element on a stack can be referenced and modified without removal by using the register deferred addressing mode. Other elements on the stack can be referenced by using displacement mode addressing.

The processor designates Register 14 as the Stack Pointer for use with both the subroutine Branch or Jump instructions, and the procedure Call instructions. On routine entry, the processor automatically saves the address of the instruction that follows the routine call on the stack. It uses the Program Counter and the Stack Pointer to perform the operation. Before entering the subroutine, the Program Counter contains the address of the instruction following the Branch, Jump, or Call instruction. The Stack Pointer contains the address of the last item on the stack. The processor pushes the contents of the Program Counter on the stack. On return from a subroutine, the processor automatically restores the Program Counter by popping the return address off the stack.

Also for the procedure Call instructions, the processor designates Register 12 as an Argument Pointer, and Register 13 as a Frame Pointer. The Argument Pointer is used to pass the address of the argument list to a called procedure, and the Frame Pointer is used to keep track of nested Call instructions.

An argument list is a formal data structure that contains the arguments required by the procedure being called. Arguments may be actual values, addresses of data structures, or addresses of other procedures. An argument list can be passed in either of two ways: by passing only its address, or by passing the entire list on the user stack. The first method is used to pass long argument lists, or lists that are to be preserved. The second method is generally used when calling procedures that do not require arguments, or when building an argument list dynamically.

When a procedure Call instruction is issued, the processor uses the Argument Pointer to pass arguments to the procedure. If arguments were passed on the stack, the processor automatically pops the arguments off on return from the procedure.

The importance of the way the Call instructions work is that nested calls can be traced back to any previous level. The Call instructions always keep track of nested calls by using the Frame Pointer register. The Frame Pointer contains the address on the stack of the items
pushed on the stack during the procedure call. The set of items pushed on the stack during a procedure call is known as a call frame or stack frame. Since the previous contents of the Current Frame register are saved in each call frame, the nested frames form a linked data structure which can be unwound to any level when an error or exception condition occurs in any procedure.

**PROCESSOR STATUS WORD**

The Processor Status Word (the lower word of the Processor Status Longword) is a special processor register that a program uses to check its status and to control synchronous error conditions. The Processor Status Word, contains two sets of bit fields:

- the condition codes
- the trap enable flags

The condition codes indicate the outcome of a particular logical or arithmetic operation. For example, the Subtract instruction sets the Negative bit if the result of the subtraction operation produced a negative number, and it sets the Zero bit if the result produced zero. The Branch on Condition instructions can then be used to transfer control to a code sequence that handles the condition.

There are two kinds of traps that concern the user process: trace traps and arithmetic traps. The trace trap is used by debugging programs or performance evaluators. Arithmetic traps include:

- integer, floating point, or decimal string overflow, in which the result was too large to be stored in the given format
- integer, floating point, or decimal string divide by zero, in which the divisor supplied was zero
- floating point underflow, in which the result was too small to be expressed in the given format

Of the arithmetic traps, one of two methods can be used to handle integer overflow, 1) floating underflow, and 2) decimal string overflow. If the trap enable bits are cleared in the Processor Status Word, the processor will ignore integer and decimal string overflow and floating underflow. If these conditions are to be checked, the user may check the condition codes directly (using the Branch On Condition instructions), or enable the trap bits. If the trap bits are enabled, the processor treats integer and decimal string overflow and floating underflow as exceptions (refer to Chapter 12, Exceptions). In any case, floating overflow and divide by zero traps are always enabled.

**Condition Codes**

A user program can test the outcome of an arithmetic or logical operation. The processor provides a set of condition codes and branch
instructions for this purpose. The condition codes indicate whether the previous arithmetic or logical operation produced a negative or zero result, whether there was a carry or borrow, or whether an overflow occurred. There are a variety of branch on condition instructions: those for overflow and carry or borrow, and those for signed and unsigned relational tests.

**INSTRUCTION FORMAT**

A native mode instruction has a variable-length format, and instructions may start on any byte boundary. A variable-length format not only makes code more compact, it means that the instruction set can be extended easily. The opcode for the operation is a single byte for presently existing instructions, and it is followed by zero to six operand specifiers, depending on the instruction. An operand specifier can be one to ten bytes long, depending on the addressing mode (for greater detail, refer to Chapter 5, Instruction Formats and Addressing Modes). Figure 2-2 illustrates the autodecrement mode move long instruction as a string of bytes starting with the opcode followed by two operand specifiers. The assumed starting location in this example is 00003000. When the processor completes the execution of an instruction, the Program Counter contains the address of the first byte of the next instruction. The Program Counter operation is totally transparent to the programmer.

```
MACHINE CODE: (ASSUMED STARTING LOCATION 00003000)
00003000  D0  OPCODE FOR MOVE LONG INSTRUCTION
00003001  73  AUTODECREMENT MODE, REGISTER R3
00003002  54  REGISTER MODE, REGISTER R4
```

Figure 2-2  Autodecrement Move Long Instruction

The Program Counter itself can be used to identify operands. The assembler translates many types of operand references into addressing modes using the Program Counter. Autoincrement mode using the Program Counter, which is also called immediate mode, is used to specify in-line constants other than those available with literal mode addressing. Autoincrement deferred mode using the Program Counter, or absolute mode, is used to reference an absolute address. Displacement and displacement deferred modes using the Program Counter are used to specify an operand using an offset from the current location.
Addressing using the Program Counter enables the coding of position independent code. Position independent code can be executed anywhere in virtual address space after it has been linked, since program linkages can be identified as absolute locations in virtual address space and all other addresses can be identified relative to the current instruction.

INSTRUCTION SET
At any one time, the processor executes one of two instruction sets: native mode or compatibility mode. In native mode the processor executes a large set of variable-length instructions, recognizes a variety of data types, and uses sixteen 32-bit general purpose registers. In compatibility mode the processor executes a set of PDP-11 instructions, recognizes integer data, and uses eight 16-bit general purpose registers. While native mode is the primary instruction execution state of the machine and compatibility mode is the secondary state, their instruction sets are closely related, and their programming characteristics are very similar. A user process can consist of either a native mode image or a compatibility mode image.

A native instruction consists of an operation code (opcode) and zero or more operands, which are described by data type and access mode. The native instruction set consists of over 240 different instructions. Most operations can be applied to more than one type of data, which can be addressed with any addressing mode. Thus, the native instruction set offers a wide variety of instructions from which to choose.

In spite of the large number of instructions, the native instruction set is a natural programming language that is very easy to learn. Many of the instructions correspond directly to high-level language statements, and the assembler mnemonics are readily associated with the instruction function.

To choose the appropriate instruction, it is only necessary to become familiar with the operations, data types, and addressing modes. For example, the ADD operation can be applied to any of several sizes of integer, floating point, or packed decimal operands, and each operand can be addressed directly in a register (except packed decimal), directly in memory, or indirectly through pointers stored in registers or memory locations.

NATIVE INSTRUCTION SET
The instruction set that the processor executes is selected under operating system control to either native mode or compatibility mode. The native mode instruction set is based on over 240 different opcodes.
The opcodes can be grouped into classes based on their function and use. Instructions used to manipulate the general data types include:

- integer and floating point instructions
- packed decimal instructions
- character string instructions
- bit field instructions

Instructions that are used to manipulate special kinds of data include:

- queue manipulation instructions
- address manipulation instructions
- general register manipulation instructions

Instructions that provide basic program flow control, and enable the user to call procedures are:

- branch, jump and case instructions
- subroutine call instructions
- procedure call instructions

**Integer and Floating Point Instructions**

The logical and arithmetic processor instructions illustrate how the opcodes, data types, and addressing modes can be combined in an instruction. Most of the operations provided for integer data are also provided for floating point and packed decimal data. Exceptions are the strictly logical operations for integer data (such as bit clear, bit set, complement), the multiword arithmetic instructions for integer data (such as Add/Subtract with Carry and Extended Multiply and Extended Divide), and the Extended Modulus and Polynomial instructions for floating point data.

The arithmetic instructions include both 2-operand and 3-operand forms that eliminate the need to move data to and from temporary operands. The 2-operand instructions store the result in one of the two operands, as in “Set A equal to A plus B.” The 3-operand instructions effectively implement the high-level language statements in which two different variables are used to calculate a third, such as “Set C equal to A plus B.” The 3-operand instructions are applicable to both integer and floating point data, and equivalent instructions exist for packed decimal data.

To illustrate the instruction set and addressing modes, consider the FORTRAN language statement:

\[ A(I) = B(I) \times C(I) \]

where A, B, and C are statically allocated REAL*4 arrays and I is
INTEGER*4. A code sequence that performs this operation is:

```
MOVL  I,R0  ;Move the longword I
to a register
MULF3 B[R0],C[R0],A[R0]  ;3-operand floating
    ;multiply
```

The Extended Divide (EDIV) instruction divides a quadword integer by a longword and produces a longword quotient and a longword remainder.

The Extended Modulus (EMOD) instructions multiply a floating point number with an extended precision floating point number (extended by eight bits for an effective 9 or 19 digits of accuracy) and returns the integer portion and the fractional portion separately. This instruction is particularly useful for preserving the precision of input throughout trigonometric and exponential function evaluation.

The Polynomial Evaluation (POLY) instructions evaluate a polynomial from a table of coefficients using Horner's method. This instruction is used extensively in the high-level languages' math library for operations such as sine and cosine.

**Packed Decimal Instructions**

Many of the operations for integer and floating point data also apply to packed decimal strings. They include:

- **Move Packed (MOVP)** for copying a packed decimal string from one location to another, and **Arithmetic Shift Packed (ASHP)** for scaling a packed decimal up or down by a given power of 10 while moving it, and optionally rounding the value.

- **Compare Packed (CMPP)** for comparing two packed decimal strings. **Compare Packed** has two variations: a 3-operand (CMPP3) instruction for strings of equal length, and a 4-operand instruction (CMPP4) for strings of differing lengths.

- **Convert Instructions. These instructions enable conversion between packed decimal format and commonly used numeric formats. Numeric with trailing sign allows various sign encodings including zoned and overpunched.**

- **Add Packed (ADDP) and Subtract Packed (SUBP)** for adding or subtracting two packed decimal strings, with the option of replacing the addend or subtrahend with the result (ADDP4 and SUBP4), or storing the result in a third string (ADDP6 or SUBP6).

- **Multiply Packed (MULP) and Divide Packed (DIVP)** for multiplying or dividing two packed decimal strings and storing the result in a third string.
In addition, the packed decimal instructions include a special packed decimal string to character string conversion instruction that provides output formatting: the Edit instruction.

**Edit Instruction**
The Edit Packed to Character String (EDITPC) instruction supplies formatted numeric output functions. The instruction converts a given packed decimal string to a character string using selected pattern operators. The pattern operators enable the creation of numeric output fields with any of the following characteristics:
- leading zero fill
- leading zero protection
- leading asterisk fill protection
- a floating sign
- a floating currency symbol
- special sign representations
- insertion characters
- blank when zero

**Character String Instructions**
The character string instructions operate on strings of bytes. They include:
- move string instructions, with translation options
- string compare instructions
- single character search instructions
- substring search instructions

There are two basic forms of Move instructions for character strings. The Move Character instructions (MOVC3 and MOVC5) simply copy character strings from one location to another. They are optimized for block transfer operations. The 5-operand variation enables the user to supply a fill character that the instruction uses to pad out the destination location to a given size.

The Move Translated Characters (MOVTC) and Move Translated Until Character (MOVTUC) instructions actually create new character strings using a translation table. The Compare Characters (CMPC) instructions provide character-by-character byte string compares. The Locate Character (LOCC) and Skip Character (SKPC) instructions are search instructions for single characters within a string. The Match Characters (MATCHC) instruction is similar to the Locate Character instruction, but it locates multiple-character substrings. MATCHC searches a string for the first occurrence of a given substring. The
Span Characters (SPANC) and Scan Characters (SCANC) instructions are search instructions that look for members of character classes.

The 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.

Variable-Length Bit Field Instructions
The bit field instructions enable the user to define, access, and modify fields whose size and location are user specified. Location is determined from a base address or a register and a signed bit offset. If the field is in memory, the offset range can be as large as $2^{32} - 1$ bits (approximately 16 million bytes). If the field is in a register, the offset can be large as 31. Fields of arbitrary lengths (0 to 32 bits) can be used for storing data structure header information compactly, for status codes, or for creating user data types. The field instructions enable the user to manipulate fields easily.

The Insert Field and Extract Field instructions store data in and retrieve data from fields. The Compare Field and Find First instructions enable the user to test the contents of a field. Compare Field extracts a field and then compares it with a given longword. The Find First instructions locate the first bit in a field that is clear (FFC) or set (FFS), scanning from low-order bit to high-order bit.

Queue Instructions
The processor has six instructions that allow easy construction and maintenance of queue data structures. Queues manipulated using the queue instructions are circular, doubly linked lists of data items.

The first longword of a queue entry contains the forward pointer to the next entry in the queue, and the next longword contains the backward pointer to the preceding entry in the queue.

Two types of queues are provided: absolute and self-relative. Absolute queues use pointers that are virtual addresses, whereas self-relative queues use pointers that are relative displacements (for greater detail, refer to Chapter 4, Data Types).

Address Manipulation Instructions
Because the processor offers a variety of addressing modes that
enable easy access to data structures by keeping base addresses and indices in registers, address manipulation is frequent. The processor provides two instructions that enable the fetching of an address without actually accessing the data at that location:

- the Move Address (MOVA) instruction, which stores the address of a byte, word, longword (and floating), or quadword (and double floating) datum in a specified register or location in memory.
- the Push Address (PUSHA) instruction, which stores the address of a byte, word, longword (and floating), or quadword (and double floating) datum on the stack.

**General Register Manipulation Instructions**

The general register manipulation instructions enable any user program to save or load the general purpose registers in one operation, examine the Processor Status Longword, and set or clear status bits in the Processor Status Word.

**Branch, Jump and Case 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, a displacement (offset) is added to the current contents of the Program Counter to obtain the new address. With jump instructions, the user specified address is 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 and many conditional branch instructions. Conditional branch instructions include:

- branch on bit instructions
- set and clear bit instructions with a branch if it is already set or cleared
- loop instructions that increment or decrement a counter, compare it with a limit value, and branch on a relational condition
- computed branch instruction in which a branch may take place to one of several locations depending on a computed value

The Branch on Condition (B) instructions enable the transfer of control to another location depending on the status of one or more of the condition codes in the Processor Status Word (PSW). There are three groups of Branch on Condition instructions:
VAX-11 Architecture Overview

- the signed relational branches, which are used to test the outcome of instructions operating on integer and field data types being treated as signed integers, floating point data types, and decimal strings
- the unsigned relational branches, which are used to test the outcome of instructions operating on integer and field data types being treated as unsigned integers, character strings, and addresses
- the overflow and carry test branches, which are used for checking overflow when traps are not enabled, for multiprecision arithmetic, and for the results of special instructions

There are also general purpose Branch on Bit instructions similar to Branch on Condition. The Branch on Low Bit Set (BLBS) and Branch on Low Bit Clear (BLBC) instructions test bit 0 of an operand, which is useful for testing Boolean values. The Branch on Bit Set (BBS) and Branch on Bit Clear (BBC) instructions test any selected bit.

There are special kinds of Branch on Bit instructions that are actually bit set/clear instructions. The Branch on Bit Set and Set (BBSS) is an example. The instruction branches if the indicated bit is set, otherwise it falls through. In either case, the instruction sets the given bit. The BBSS instruction can thus be thought of as a Bit Set instruction with a branch side-effect if the bit was already set. There are four permutations:
- Branch on Bit Set and Set (BBSS)
- Branch on Bit Clear and Clear (BBCC)
- Branch on Bit Set and Clear (BBSC)
- Branch on Bit Clear and Set (BBCS)

These instructions are particularly useful for keeping track of procedure completion or initialization, and for signaling the completion or initialization of a procedure to a cooperating process. In addition, there are two Branch on Bit Interlocked instructions that provide control variable protection:
- Branch on Bit Set and Set Interlocked (BBSSI)
- Branch on Bit Clear and Clear Interlocked (BBCCI)

The processor provides a memory interlock on these instructions. No other BBSSI or BBCCI operation can interrupt these instructions to gain access to the byte containing the control variable between the testing of the bit and the setting or clearing of the bit.

The processor offers three types of branch instructions that can be used to write efficient loops. The first type provides the basic subtract-one-and-branch loop. The counterpart to subtract-one-and-branch is add-one-and-branch. The third type of loop instruction efficiently im-
implements the FORTRAN language DO statement and the BASIC language FOR statement: Add Compare and Branch (ACB).

The processor also provides a branch instruction that implements higher-level language computed GO TO statements: the CASE instruction. The CASE instruction requires as input a list of displacements that generate different branch addresses indexed by the value obtained as a selector. The branch falls through if the selector does not fall within the limits of the list.

**Subroutine Branch, Jump, and Return Instructions**

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. A Branch to Subroutine instruction requires either a byte (BSBB) or word (BSBW) displacement. With Jump to Subroutine, regular addressing is used.

The subroutine call instructions are 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.

**Procedure Call and Return Instructions**

Procedures are general purpose routines that use argument lists passed automatically by the processor. The procedure Call instructions enable language processors and the operating system to provide a standard calling interface. 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
- initialize the arithmetic trap enables to a given state

When a Call procedure instruction is issued, the address of the procedure called is required. The first word of a procedure contains an entry mask that is used in the same way as the entry mask defined for the Push Registers instruction. Each set bit of the 12 low-order bits in the word represents one of the general registers, R0 through R11, that the procedure uses. The Call instruction examines this word and saves the indicated registers on the stack. In addition, the Call instruction automatically saves the contents of the Frame Pointer, Argument Pointer, and Program Counter registers. This is an extremely efficient
way to ensure that registers are saved across procedure calls. No general register is saved that does not have to be saved.

The Call Procedure with General Argument List (CALLG) instruction accepts the address of an argument list and passes the address to the procedure in the Argument Pointer register. The Call Procedure with Stack Argument List (CALLS) passes the argument list placed on the stack by the user, if any, by loading the Argument Pointer register with its stack address.

When a procedure completes execution, it issues the Return from Procedure instruction (RET). Return uses the Frame Pointer register to find the saved registers that it restores, and to clean up any data left on the stack, including nested routine linkages. A procedure can return values using the argument list or other registers.

Miscellaneous Special Purpose Instructions
The processor has a number of special purpose instructions. They include:

- Cyclic Redundancy Check (CRC)
- Breakpoint Fault (BPT)
- Extended Function Call (XFC)
- No Operation (NOP)
- Halt

The Cyclic Redundancy Check (CRC) instruction calculates a cyclic redundancy check for a given string using any CRC polynomial up to 32 bits long. The user supplies the string for the CRC, and a table for the CRC function. The operating system library includes tables for standard CRC functions, such as CRC-16.

The Breakpoint Fault (BPT) instruction makes the processor execute the kernel mode condition handler associated with the Breakpoint Fault exception vector. BPT is used by the operating system debugging utilities, but can also be used by any process that sets up a Breakpoint Fault condition handler.

The Extended Function Call (XFC) instruction allows escapes to customer-defined instructions in writable control store. The NOP instruction is useful for debugging. The HALT instruction is a privileged instruction issued only by the operating system to halt the processor when bringing the system down by operator request.

SEPARATION OF PROCEDURE AND DATA
The VAX-11 architecture encourages (and provides the mechanisms to facilitate) separation of procedure (instructions) and writable data. Procedures may not write data which is to be subsequently executed
as an instruction without an intervening REI instruction being executed (refer to Exceptions, Chapter 12), or an intervening context switch occurring (Refer to Process Structure Chapter in appropriate Hardware Handbook). If no REI or context switch occurs between a procedure writing data as instructions to be executed, and those instructions being executed, the instructions executed are UNPREDICTABLE.

EXCEPTIONS
There are some situations in which the outcome of an operation need not be tested. The processor recognizes events that require testing, and automatically changes the normal flow of the program when they occur. These events, called exceptions, are the direct result of executing a specific instruction. Exceptions also include errors automatically detected by the processor, such as improperly formed instructions.

All exceptions trap to operating system software. There are essentially no fatal exceptions. All exceptions either wait for the instruction that caused them to complete before trapping or they restore the processor to the state it was in just prior to executing the instruction that caused the exception. In either case, the instruction can be retried after the cause of the exception is cleared. Depending on the exception, it may be desirable to correct the situation and continue. If not, the operating system issues an appropriate message and aborts the instruction stream in progress. To continue, the operating system software can be requested to start execution of a condition handler automatically when an exception occurs.

Handling Exceptions
When an exception occurs, the processor immediately saves the current state of execution and traps to the operating system. The operating system automatically searches for a procedure that can handle the exception. Procedures that respond to exceptions are called condition handlers. The user can declare a condition handler for an entire image and for each individual procedure called. Because the processor keeps track of nested calls using the Frame Pointer register, it is possible to declare condition handlers for procedures that call other procedures in which exceptions might occur. In this case, the operating system automatically traces back through the call frames to locate a condition handler capable of handling the appropriate exception condition.

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

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

PROCESSING CONCEPTS FOR SYSTEM PROGRAMMING
The VAX-11 processor is specifically designed to support a high-performance multiprogramming environment. The major advantage of a multiprogramming system is its ability to utilize most efficiently those resources of the computer that are being shared by several executing environments. For example, multiprogramming enables the execution of many application systems, and the interactive development of application programming simultaneously. The characteristics of the hardware system that support multiprogramming are:

- rapid context switching
- priority dispatching
- virtual addressing and memory management

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

Context Switching
In a multiprogramming environment, several individual streams of code can be ready to execute at any one time. Instead of allowing each stream to execute to completion serially (as in a batch-only stream), the operating system can intervene and switch between the streams of code which are ready to execute.

To support multiprogramming for high-performance system, the processor enables the operating system to switch rapidly between individual streams of code. The stream of code the processor is executing at any one time is determined by its hardware context. Hardware context includes the information loaded in the processor's registers that identifies:
- location of stream's instructions and data
- which instruction to execute next
- processor status during execution

A process is a stream of instructions and data defined by a hardware context. Each process has a unique identification in the stream. The operating system switches between processes by requesting the processor to save one process hardware context and load another. Context switching occurs rapidly because the processor instruction set includes save hardware context and load hardware context instructions. The operating system's context switching software does not have to individually save or load the processor registers which define the hardware context.

**Priority Dispatching**
While running in the context of one process, the processor executes instructions and controls data flow to and from peripherals and main memory. To share processor, memory and peripheral resources among many processes, the processor provides two arbitration mechanisms that support high-performance multiprogramming: exceptions and interrupts. Exceptions are events that occur synchronously with respect to instruction execution, while interrupts are external events that occur asynchronously.

The flow of execution can change at any time, and the processor distinguishes between changes in flow that are local to a process and those that are system-wide. Process-local changes occur as the result of a user software error or when user software calls operating system services. Process-local changes in program flow are handled through the processor's exception detection mechanism and the operating system's exception dispatcher.

System-wide changes in flow generally occur as the result of interrupts from devices or interrupts generated by the operating system software. Interrupts are handled by the processor's interrupt detection mechanism and the operating system's interrupt service routines. (System-wide changes in flow may also occur as the result of severe hardware errors, in which case they are handled either as special exceptions or high-priority interrupts.)

System-wide changes in flow take priority over process-local changes in flow. Furthermore, the processor uses a priority system for servicing interrupts. To arbitrate between all possible interrupts, each kind of interrupt is assigned a priority, and the processor responds to the highest priority interrupt pending. For example, interrupts from the high-speed disk devices take precedence over interrupts from low-speed devices.
The processor services interrupts between instructions, or at well-defined points during the execution of long, iterative instructions. When the processor acknowledges an interrupt, it switches rapidly to a special system-wide context to enable the operating system to service the interrupt. System-wide changes in the flow of execution are handled in such a way as to be totally transparent to individual processes.

**Virtual Addressing and Virtual Memory**

The processor's memory management hardware enables the operating system to provide an execution environment that allows users to write programs without having to know where the programs are loaded in physical memory, and to write programs that are too large to fit in the physical memory they are allocated.

The processor provides the operating system with the ability to provide virtual addressing. A virtual address is a 32-bit integer that a program uses to identify storage locations in virtual memory. Virtual memory is the set of all physical memory locations in the system plus the set of disk blocks that the operating system designates as extensions to physical memory.

A physical address is an address that the processor uses to identify physical memory storage locations and peripheral controller registers. It is the address that the processor sends out on the SBI bus to which the memory and peripheral adaptors respond.

The processor must be able to translate the virtual addresses provided by the programs it executes into the physical addresses recognized by the memory and peripherals. To provide virtual to physical address mapping, the processor has address mapping registers controlled by the operating system and an integrated address translation buffer.

The mapping registers enable the operating system to relocate programs in physical memory, to protect programs from each other, and share instructions and data between programs transparently or at their request. The address translation buffer ensures that the virtual address to physical address translation takes place rapidly.

**SYSTEM PROGRAMMING ENVIRONMENT**

Within the context of one process, user-level software controls its execution using the instruction sets, the general registers and the Processor Status Word. Within the multiprogramming environment, the operating system controls the system's execution using a set of special instructions, the Processor Status Longword, and the internal processor registers.
Processor Status Longword
A processor register called the Processor Status Longword (PSL) determines the execution state of the processor at any time. The low-order 16 bits of the Processor Status Longword are the Processor Status Word available to the user process. The high-order 16 bits provide privileged control of the system.

The fields can be grouped together by functions that control:
- the instruction set the processor is executing
- the access mode of the current instruction
- interrupt processing

The instruction set the processor executes is controlled by the compatibility mode bit in the Processor Status Longword. This bit is normally set or cleared by the operating system.

The following paragraphs discuss access modes, the native instructions primarily used by the operating system, memory management, and interrupt processing.

Processor Access Modes
In a high-performance multiprogramming system, the processor must provide the basis for protection and sharing among the processes competing for the system's resources. The basis for protection in this system is the processor's access mode. The access mode in which the processor executes determines:
- instruction execution privileges: what instructions the processor will execute
- memory access privileges: which locations in memory the current instruction can access

At any one time, the processor is executing code in the context of a particular process, or it is executing in the system-wide interrupt service context. In the context of a process, the processor recognizes four access modes: kernel, executive, supervisor, and user. Kernel is the most privileged mode and user the least privileged.

The processor spends most of its time executing in user mode in the context of one process or another. When user software needs the services of the operating system, whether for acquisition of a resource, for I/O processing, or for information, it calls those services.

The processor executes those services in the same or one of the more privileged access modes within the context of that process. That is, all four access modes exist within the same virtual address space. Each access mode has its own stack in the control region of per-process
space, and therefore each process has four stacks: one for each access mode. Note that this makes it easy for the operating system to context switch a process even when it is executing an operating system service procedure.

In any mode except kernel, the processor will not execute the instructions that:

- halt the processor
- load and save process context
- access the internal processor registers that control memory management, interrupt processing, the processor console, or the processor clock

These instructions are privileged instructions that are generally reserved to the operating system.

In any mode, the processor will not allow the current instruction to access memory unless the mode is privileged to do so. The ability to execute code in one of the more privileged modes is granted by the system manager and controlled by the operating system. The memory protection the privilege affords is enforced by the processor. In general, code executing in one mode can protect itself and any portion of its data structures from read and/or write access by code executing in any less privileged mode. For example, code executing in executive mode can protect its data structures from code executing in supervisor or user mode. Code executing in supervisor mode can protect its data structures from access by code executing in user mode. This memory protection mechanism provides the basis for system data structure integrity.

Protected and Privileged Instructions
The processor provides three types of instructions that enable user mode software to obtain operating system services without jeopardizing the integrity of the system. They include:

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

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

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

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

The operating system's privileged service procedures and interrupt and exception service routines exit using the Return from Exception or Interrupt (REI) instruction. REI is the only way in which the privilege of the processor's access mode can be decreased. Like the procedure and subroutine return instructions, REI restores the Program Counter and the processor state to resume the process at the point where it was interrupted.

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

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

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

**Memory Management**

The processor is responsible for enforcing memory protection between access modes. Memory protection, however, is only a part of the processor's memory management function. In particular, the memory management hardware enables the operating system to provide an extremely flexible and efficient virtual memory programming environment. Virtual and physical address space definitions provide the basis for the virtual memory available on a system. Virtual address space consists of all possible 32-bit addresses that can be exchanged between a program and the processor to identify a byte location in physical memory. The memory management hardware translates a virtual address into a physical address (30 bits in the case of 11/780). A physical address is the address exchanged between the processor, memory, and the peripheral adapters. Physical address space is the set of all possible physical addresses the processor can use to express unique memory locations and peripheral control registers. The bus structure and physical address space for the various VAX processors is implementation specific.

Half of the addresses in physical address space can be used to refer to real memory locations and the other half can be used to refer to peripheral device control and data registers. For example, on the 11/780, all of the the lowest-addressed half of physical address space is called memory space, and the highest-addressed half I/O space.

The following section describes the way in which the memory management hardware enables the operating system to map virtual addresses into physical addresses to provide the virtual memory available to a process.

**Virtual to Physical Page Mapping**

Virtual address space is divided into pages, where a page represents 512 bytes of contiguously addressed memory. The first page begins at byte zero and continues to byte 511. The next page begins at byte 512
and continues to byte 1023, and so forth. If we listed the first 8 pages of virtual address space, their addresses in both decimal and hexadecimal are:

<table>
<thead>
<tr>
<th>PAGE</th>
<th>ADDRESS(10)</th>
<th>ADDRESS(16)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0000-0511</td>
<td>0000-01FF</td>
</tr>
<tr>
<td>1</td>
<td>0512-1023</td>
<td>0200-03FF</td>
</tr>
<tr>
<td>2</td>
<td>1024-1535</td>
<td>0400-05FF</td>
</tr>
<tr>
<td>3</td>
<td>1536-2047</td>
<td>0600-07FF</td>
</tr>
<tr>
<td>4</td>
<td>2048-2559</td>
<td>0800-09FF</td>
</tr>
<tr>
<td>5</td>
<td>2560-3071</td>
<td>0A00-0BFF</td>
</tr>
<tr>
<td>6</td>
<td>3072-3583</td>
<td>0C00-0DFF</td>
</tr>
<tr>
<td>7</td>
<td>3584-4095</td>
<td>0E00-0FFF</td>
</tr>
</tbody>
</table>

The size of a virtual page exactly corresponds to the size of a physical page of memory, and the size of a block on disk.

To make memory mapping efficient, the processor must be able to translate virtual addresses to physical addresses rapidly. Two features providing rapid address translation are the processor's internal address translation buffer, and the translation algorithm itself.

Figure 2-3 compares the virtual and physical address format. The high-order two bits of a virtual address immediately identify the region to which the virtual address refers. Whether the address is physical or virtual, the byte within the page is the same. Thus, the processor has to know only which virtual pages correspond to which physical pages.

![Figure 2-3 Virtual and Physical Addresses](image)
The processor has three pairs of page mapping registers, one pair for each of the three regions actively used. The operating system's memory management software loads each pair of registers with the base address and length of data structures it sets up called page tables. The page tables provide the mapping information for each virtual page in the system. There is one page table for each of the three regions.

A page table is a virtually contiguous array of page table entries. Each page table entry is a longword representing the physical mapping for one virtual page. To translate a virtual address to a physical address, therefore, the processor simply uses the virtual page number as an index into the page table from the given page table base address. Each translation is good for 512 virtual addresses since the byte within the virtual page corresponds to the byte within the physical page.

Figure 2-4 shows the format of a page table entry. The high-order bits are used to indicate the page's status and protection. The page's protection can be set to prevent read and/or write access by any mode (kernel, executive, supervisor, or user). The page's status indicates what the remainder of the page table entry means. It may be, for example, a page address in physical address space, a disk sector, or a temporary pointer to a page shared by two or more processes. The system's virtual memory is a dynamic memory that is defined by the physical memory and disk pages that are virtually mapped by page table entries.

![Page Table Entry Diagram]

Figure 2-4  Page Table Entry

The operating system's memory management software maintains the page table entry protection and status bits, with the exception of the modified page bit. The processor sets the modified page bit to indicate that it has written into a physical page in memory. This is used to keep disk I/O to a minimum when paging a process.

The processor uses the page table base registers to locate the page tables, and uses the length registers as a validity check to ensure that
any given virtual page is in the range of defined page table entries. Figure 2-5 summarizes and compares the page table structures.

**SYSTEM REGION PAGE TABLE**

<table>
<thead>
<tr>
<th>Page Table Entry for Virtual Page 0 (first entry)</th>
</tr>
</thead>
<tbody>
<tr>
<td>PTE for VPN 1</td>
</tr>
<tr>
<td>PTE for VPN 2</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>Page Table Entry for Virtual Page N - 1 (last entry)</td>
</tr>
</tbody>
</table>

**PER-PROCESS PAGE TABLES**

**PROGRAM REGION PAGE TABLE**

<table>
<thead>
<tr>
<th>Page Table Entry for Virtual Page 0 (first entry)</th>
</tr>
</thead>
<tbody>
<tr>
<td>PTE for VPN 1</td>
</tr>
<tr>
<td>PTE for VPN 2</td>
</tr>
<tr>
<td>PTE for VPN 3</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>PTE for Virtual Page N-1 (last entry)</td>
</tr>
</tbody>
</table>

**CONTROL REGION PAGE TABLE**

<table>
<thead>
<tr>
<th>Page Table Entry for Virtual Page 2^22-N</th>
</tr>
</thead>
<tbody>
<tr>
<td>PTE for VPN 2^22-(N-1)</td>
</tr>
<tr>
<td>PTE for VPN 2^22-(N-2)</td>
</tr>
<tr>
<td>PTE for VPN 2^22-(N-3)</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>PTE for VPN 2^22-1 (last entry)</td>
</tr>
</tbody>
</table>

Figure 2-5  Page Table Structure

All process page tables have virtual addresses in the system region of virtual address space, but the system region page table is located by its address in physical memory. That is, the system region page table base register contains the physical address of the page table base, while the process page table base registers contain the virtual addresses of their page table bases. Because a per-process page table entry is referred to by its virtual address, the hardware translates its virtual address using the system page table.

There are two advantages to using a virtual address as the base address of a per-process page table. The first advantage is that all page tables do not have to reside in physical memory. The system region page table is the only page table that needs to be resident in physical memory. All process page tables can reside on disk; that is, process page tables can themselves be paged and swapped as necessary.
The second advantage is that the operating system's memory management software can allocate per-process page tables dynamically, because the per-process page tables do not need to be mapped into contiguous physical pages. And although the system region page table must be mapped into contiguous physical pages, this requirement does not restrict physical memory allocation. The region is shared among processes, and therefore does not require redefinition from context to context.

To illustrate the efficiency of this memory mapping scheme, suppose that 16 processes, each of which is using 4 million bytes of virtual address space, are known to the system at the same time (for a total of 64 Mb of virtual address space). One system page table entry maps one page of per-process page table entries, and one page of per-process page table entries maps 65,536 (64K) bytes of virtual address space (since it is possible to store 128 page table entries in a single page of memory). Therefore one page of system page table maps 128 pages of per-process page tables, which in turn maps 8 Mb of process virtual address space. Thus the system region page table needed to map these 16 processes requires approximately 8 physical pages (4K bytes) of memory.

Exception and Interrupt Vectors
The processor can automatically initiate changes in the normal flow of program execution. The processor recognizes two kinds of events that cause it to invoke conditional software: exceptions and interrupts. Some exceptions affect an individual process only, such as arithmetic traps, while others affect the system as a whole, for example, machine check. Interrupts include both device interrupts, such as those signaling I/O completion, and software-requested interrupts, such as those signaling the need for a context switch operation.

The processor knows which software to invoke when an exception or interrupt occurs because it references specific locations, called vectors, to obtain the starting address of the exception or interrupt dispatcher. The processor has one internal register, the System Control Block Base Register, that the operating system loads with the physical address of the base of the System Control Block, which contains the exception and interrupt vectors. The processor locates each vector by using a specific offset into the System Control Block. Figure 2-6 illustrates the vectors in the System Control Block. Each vector tells the processor how to service the event, and contains the system region virtual address of the routine to execute. Note that vector 14 (hex) can be used as a trap to writable control store to execute user-defined instructions, and the vector contains information passed to microcode.
## Exception Vectors

<table>
<thead>
<tr>
<th>Vector</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>Machine Check</td>
</tr>
<tr>
<td>8</td>
<td>Kernel Stack Not Valid</td>
</tr>
<tr>
<td>C</td>
<td>Power Fail</td>
</tr>
<tr>
<td>10</td>
<td>Reserved or Privileged Instruction</td>
</tr>
<tr>
<td>14</td>
<td>Customer Reserved Instruction</td>
</tr>
<tr>
<td>18</td>
<td>Reserved or IllegalOperand</td>
</tr>
<tr>
<td>1C</td>
<td>Reserved or Illegal Addressing Mode</td>
</tr>
<tr>
<td>20</td>
<td>Access Violation</td>
</tr>
<tr>
<td>24</td>
<td>Translation Not Valid (page fault)</td>
</tr>
<tr>
<td>28</td>
<td>Trace Trap</td>
</tr>
<tr>
<td>2C</td>
<td>Breakpoint Trap</td>
</tr>
<tr>
<td>30</td>
<td>Compatibility Mode Trap</td>
</tr>
<tr>
<td>34</td>
<td>Arithmetic Trap</td>
</tr>
<tr>
<td>40</td>
<td>Change Mode to Kernel</td>
</tr>
<tr>
<td>44</td>
<td>Change Mode to Executive</td>
</tr>
<tr>
<td>48</td>
<td>Change Mode to Supervisor</td>
</tr>
<tr>
<td>4C</td>
<td>Change Mode to User</td>
</tr>
</tbody>
</table>

## Interrupt Vectors

<table>
<thead>
<tr>
<th>Vector</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>84</td>
<td>Software Level 1</td>
</tr>
<tr>
<td>88</td>
<td>Software Level 2</td>
</tr>
<tr>
<td>BF</td>
<td>Software Level F</td>
</tr>
<tr>
<td>CO</td>
<td>Interval Timer</td>
</tr>
<tr>
<td>100</td>
<td>Device Level 14, device 0</td>
</tr>
<tr>
<td>101</td>
<td>Device Level 14, device 1</td>
</tr>
<tr>
<td>13F</td>
<td>Device Level 14, device 15</td>
</tr>
<tr>
<td>140</td>
<td>Device Level 15, device 0</td>
</tr>
<tr>
<td>17F</td>
<td>Device Level 15, device 15</td>
</tr>
<tr>
<td>180</td>
<td>Device Level 16, device 0</td>
</tr>
<tr>
<td>1BF</td>
<td>Device Level 16, device 15</td>
</tr>
<tr>
<td>1C0</td>
<td>Device Level 17, device 0</td>
</tr>
<tr>
<td>1FF</td>
<td>Device Level 17, device 15</td>
</tr>
</tbody>
</table>

Offset from System Control Block Base Register (HEX)

Figure 2-6 System Control Block
Interrupt Priority Levels
Exceptions do not require arbitration since they occur synchronously with respect to instruction execution. Interrupts, on the other hand, can occur at any time. To arbitrate between interrupt requests that may occur simultaneously, the processor recognizes 31 interrupt priority levels.

The highest 16 interrupt priority levels are reserved for interrupts generated by hardware, and the lowest 16 interrupt priority levels are reserved for interrupts requested by software. Normal user software runs at process level, which is interrupt priority level zero.

To handle interrupt requests, the processor enters a special system-wide context. In the system-wide context, the processor executes in kernel mode using a special stack called the interrupt stack. The interrupt stack cannot be referenced by any user mode software because the processor only selects the interrupt stack after an interrupt, and all interrupts are trapped through system vectors.

The interrupt service routine executes at the interrupt priority level of the interrupt request. When the processor receives an interrupt request at a level higher than that of the currently executing software, the processor honors the request and services the new interrupt at its priority level. When the interrupt service routine issues the REI (Return from Exception or Interrupt) instruction, the processor returns control to the previous level.

I/O Space and I/O Processing
An I/O device controller has a set of control/status and data registers. The registers are assigned addresses in physical address space, and their physical addresses are mapped, and thus protected, by the operating system’s memory management software. That portion of physical address space in which device controller registers are located is called I/O space. In the VAX-11/780 for example, I/O space occupies the highest-addressed half of physical address space, and is two bytes in length. A portion of I/O space is specifically mapped into UNIBUS addresses, and is called UNIBUS space.

No special processor instructions are needed to reference I/O space. The registers are simply treated as locations containing integer data. An I/O device driver issues commands to the peripheral controller by writing to the controller’s registers as if they were physical memory locations. The software reads the registers to obtain the controller status. The driver controls interrupt enabling and disabling on the set of controllers for which it is responsible. When interrupts are enabled, an interrupt occurs when the controller requests it. The processor accepts the interrupt request and executes the driver’s interrupt ser-
vice routine if it is not currently executing on a higher-priority interrupt level.

**Process Context**
For each process eligible to execute, the operating system creates a data structure called the software process control block. Within the software process control block is a pointer to a data structure called the hardware process control block. The hardware process control block is illustrated in Figure 2-7. It contains the hardware process context, that is, all the data needed to load the processor's programmable registers when a context switch occurs. To give control of the processor to a process, the operating system loads the processor's Process Control Block Base register with the physical address of a hardware process control block and issues the Load Process Context instruction. The processor loads the process context in one operation and is ready to execute code within that context.

As Figure 2-7 illustrates, a process control block not only contains the state of the programmable registers, it also contains the definition of the process virtual address space. Thus, the mapping of the process is automatically context-switched.

Furthermore, the process control block provides the mechanism for triggering asynchronous system traps to user processes. The Asynchronous System Trap field enables the processor to schedule a software interrupt to initiate an AST routine and ensure that they are delivered to the proper access mode for the process.

**FOR MORE INFORMATION ON VAX-11 ARCHITECTURE**
Expanded information on the VAX-11 Architecture is available in the subsequent chapters of this handbook, and the individual VAX-11 Hardware Handbooks.

Chapter 3 treats memory, registers and processor status in detail. Chapter 4 provides a detailed discussion of VAX-11 data types. Chapter 5 presents VAX-11 instruction formats and addressing modes. Chapters 6 through 11 describe the VAX-11 native mode instruction set. Chapter 12 discusses user level exception handling in the VAX-11 architecture. Chapter 13 treats the compatibility mode instruction set included in the VAX-11 architecture.

The appendices provide useful summaries and information related to the VAX-11 architecture. Finally, those aspects of the VAX-11 architecture required by operating system implementors are presently described in the respective VAX-11 Hardware Handbook.
## VAX-11 Architecture Overview

<table>
<thead>
<tr>
<th>Type of Stack Pointer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kernel mode stack pointer</td>
</tr>
<tr>
<td>Executive mode stack pointer</td>
</tr>
<tr>
<td>Supervisor mode stack pointer</td>
</tr>
<tr>
<td>User mode stack pointer</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Register</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register 0</td>
</tr>
<tr>
<td>Register 1</td>
</tr>
<tr>
<td>Register 2</td>
</tr>
<tr>
<td>Register 3</td>
</tr>
<tr>
<td>Register 4</td>
</tr>
<tr>
<td>Register 5</td>
</tr>
<tr>
<td>Register 6</td>
</tr>
<tr>
<td>Register 7</td>
</tr>
<tr>
<td>Register 8</td>
</tr>
<tr>
<td>Register 9</td>
</tr>
<tr>
<td>Register 10</td>
</tr>
<tr>
<td>Register 11</td>
</tr>
<tr>
<td>Register 12</td>
</tr>
<tr>
<td>Register 13</td>
</tr>
<tr>
<td>Register 14</td>
</tr>
<tr>
<td>Register 15</td>
</tr>
</tbody>
</table>

### Processor Status Longword

<table>
<thead>
<tr>
<th>Field</th>
</tr>
</thead>
<tbody>
<tr>
<td>Program Region Base Register</td>
</tr>
<tr>
<td>Program Region Length Register</td>
</tr>
<tr>
<td>Control Region Base Register</td>
</tr>
<tr>
<td>Control Region Length Register</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
</tr>
<tr>
<td>27</td>
</tr>
<tr>
<td>26</td>
</tr>
<tr>
<td>24</td>
</tr>
<tr>
<td>23</td>
</tr>
<tr>
<td>22</td>
</tr>
<tr>
<td>21</td>
</tr>
<tr>
<td>0</td>
</tr>
</tbody>
</table>

*Enable performance monitor

**Asynchronous System Trap pending

---

**Figure 2-7**  
Hardware Process Control Block
CHAPTER 3
MEMORY, REGISTERS AND
PROCESSOR STATUS

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, (i.e., the concurrent execution of a number of processes in a single computer system). A process can be defined as a single stream of machine instructions executed in sequence.

The virtual address space (the memory space as it appears to a process) is mapped into the physical address space (the memory space which actually exists in the hardware) by the processor's memory management logic. In addition, the memory management hardware supports paging, the technique 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 \(2^{32}\) (approximately 4.3 giga) bytes. Certain 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 detail, while subsequent chapters will describe the instructions and data which make up a VAX-11 process.

MEMORY
The memory space addressable by any program is \(2^{32}\) bytes (i.e., 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. System space contains 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 process space is independent of the others, while the system space is shared by all.

![Figure 3-1 Address Spaces in Process Context](image)

The basic addressable unit in VAX-11 is the 8-bit byte. Larger units are constructed by doubling byte sizes: 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 (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 four, 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 should be aware however, that this conversion has a serious impact on performance, and data structures should be designed in such a way that the natural alignment of operands is preserved wherever possible.

The VAX-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 explicit 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 code running in less privileged modes 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 se-
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 (i.e., the address of the first byte of a page is a multiple of 512). Virtual addresses are 32 bits long, and are divided up by the memory management logic as shown in Figure 3-2.

![Virtual Address Format](image)

Figure 3-2  Virtual Address Format

The nine low-order bits select a byte within a page, and are unchanged by the address translation process. The two high-order bits select the P0, P1, or system portion of the address space. The remaining 21 bits are used to obtain a Page Table Entry (PTE) from the P0, P1, or system page table as appropriate. Figure 3-3 illustrates the page table entry format.

![Page Table Entry Format](image)

Figure 3-3  Page Table Entry Format
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 (i.e., it identifies a page in memory, rather than one in the swapping space on a disk)
- modification flag, set by the processor whenever a write to the page occurs

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. P0PT and P1PT are to be allocated in contiguous areas of system space (i.e., virtual memory). Thus, the mapping for process space addresses involves two memory references—one to translate the process space address into a physical memory address, and the second to translate the system virtual address of the table containing the first translation. 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.

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 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; therefore, PC is not used as a temporary, accumulator, or index register.

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

**FP**
R13 is the Frame Pointer (FP). The VAX-11 procedures 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.
Memory, Register and Processor Status

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, CRC, 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. This is because R0 and R1 are used to return function values. Table 3-1 summarizes the hardware and conventional software use of the general registers.

Table 3-1  Special Register Usage

<table>
<thead>
<tr>
<th>Registers</th>
<th>Hardware Use</th>
<th>Conventional Software Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>R0</td>
<td>Results of POLY, CRC; length counter in character &amp; decimal instructions</td>
<td>Results of functions, status of services (not saved or restored on procedure call)</td>
</tr>
<tr>
<td>R1</td>
<td>Result of POLYD; address counter in character &amp; decimal instructions</td>
<td>Result of functions (not saved or restored on procedure call)</td>
</tr>
<tr>
<td>R2, R4</td>
<td>Length counter in character &amp; decimal instructions</td>
<td>Any</td>
</tr>
<tr>
<td>R3, R5</td>
<td>Address counter in character &amp; decimal instructions</td>
<td>Any</td>
</tr>
</tbody>
</table>
Memory, Register and Processor Status

<table>
<thead>
<tr>
<th>Registers</th>
<th>Hardware Use</th>
<th>Conventional Software Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>R6:R11</td>
<td>None</td>
<td>Any</td>
</tr>
<tr>
<td>AP (R12)</td>
<td>Argument pointer saved &amp; loaded by CALL, restored by RET</td>
<td>Argument pointer (base address of argument list)</td>
</tr>
<tr>
<td>FP (R13)</td>
<td>Frame pointer saved &amp; loaded by CALL, used &amp; restored by RET</td>
<td>Frame pointer; condition signalling</td>
</tr>
<tr>
<td>SP (R14)</td>
<td>Stack pointer</td>
<td>Stack pointer</td>
</tr>
<tr>
<td>PC (15)</td>
<td>Program counter</td>
<td>Program counter</td>
</tr>
</tbody>
</table>

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. The "top" of the stack is that location in the block which contains the next candidate for removal. An item is added to the stack ("pushed on") by decrementing the register which serves as the stack pointer, and storing the item at the address in the updated register. The pointer is decremented by the length of the item added to the stack, to allow enough room for it. Conversely, the top item is removed ("popped off") by adding the length of the item to the stack pointer after the last use of the item. These operations are built into the basic addressing mechanisms of VAX-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.

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

Many VAX-11 processor operations make use of the stack implicitly (i.e., without explicit specification of SP in an operand specifier). This occurs in instructions used in calling and returning from subroutines, and in the processor sequences which initiate and terminate interrupt or exception service routines. In all such cases, the processor uses the stack addressed by R14.

This does not mean that exceptions, interrupts, and system services are performed on the same 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 3-4 illustrates the relationships of the five stacks and multiple processes.

<table>
<thead>
<tr>
<th>PROCESS 1</th>
<th>PROCESS 2</th>
<th>PROCESS 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>USER 1 STACK</td>
<td>USER 2 STACK</td>
<td></td>
</tr>
<tr>
<td>SUPERVISOR 1 STACK</td>
<td>SUPERVISOR 2 STACK</td>
<td></td>
</tr>
<tr>
<td>EXEC 1 STACK</td>
<td>EXEC 2 STACK</td>
<td></td>
</tr>
<tr>
<td>KERNEL 1 STACK</td>
<td>KERNEL 2 STACK</td>
<td></td>
</tr>
<tr>
<td>INTERRUP STACK (ALL PROCESSORS)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 3-4  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-reproducible 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 pont accumulator, privileged code can still depend on it as a stack pointer, because the processor saves the floating point value and loads the pointer value when a mode change occurs.

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

**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 3-5 illustrates the Processor Status Longword.
Bits 3:0 of the PSL are termed the condition codes; in general they reflect the result status of the most recent instruction which affects them. The condition codes are tested by the conditional branch instructions.

**N Bit**—Bit 3 is the 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**—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**—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**—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 7:4 of the PSL are trap-enable flags, which cause traps to occur under special circumstances.

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

**FU Bit**—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**—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 15:8 of the PSL are unused, and reserved.

**IPL Bits**—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 of any priority. The interrupt service routine for any request, however, runs at the IPL of the request, thereby temporarily blocking interrupt requests of lower or equal priority (refer to VAX-11 Hardware Handbook for full details). Briefly, there are 31 priority levels above zero, numbered in hex 01 through 1F. Interrupt 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**—Bits 23:22 are the previous mode field, which contains the value from the current mode field at the most recent exception which transferred from a less privileged mode to this one. Previous mode is of interest, for example, in the PROBE instructions,
which enable privileged routines to determine whether a caller at the previous mode is sufficiently privileged to reference a given area of memory.

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

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

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

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

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

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

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

**CM Bit**—Bit 31 is the Compatibility Mode bit. When CM is set, the processor is in PDP-11 compatibility mode, and executes PDP-11 instructions. When CM is clear, the processor is in native mode, and executes VAX-11 instructions.
CHAPTER 4
DATA REPRESENTATION

INTRODUCTION
The VAX-11 instruction set can use a wide range of data types. The VAX data types can be separated into categories according to the groups of instructions that operate them. They are as follows:
- Integer and Floating Data Types
- Character String Data Types
- Numeric String Data Types
- Packed Decimal Data Types
- Queue Data Types
- Variable Length Bit Field Data Types
- Special Table Data Types

Figure 4-1 summarizes the VAX-11 data types. Many of these data types can be further characterized by both size and format flexibility.

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

Integer Data
VAX-11 supports integer data types of 8-, 16-, 32-, and 64-bit sizes. These are termed byte, word, longword, and quadword integers respectively. The integer data types are stored in memory in a binary format which can be treated as either signed or unsigned quantities. In the case of signed quantities, the integer is represented in 2's complement form. This means that positive numbers have a zero most significant bit (MSB) and the representation of a negative number is one greater than the bit-by-bit complement of its positive counterpart. This means that the MSB is always zero for positive values and one for negative values. When treated as unsigned quantities, integers extend upward from 0.
Figure 4-1  VAX-11 Data Types
Data Representation

- **byte**

A byte is eight contiguous bits starting on an addressable byte boundary or located in a register \( R_n <7:0> \). The bits are numbered from the right 0 through 7. Figure 4-2 illustrates the byte format.

![Figure 4-2 Byte Format](image)

A byte is specified by its address \( A \). When interpreted as a signed quantity, a byte is a 2's complement integer with bits increasing in significance from 0 through 6, and with bit 7 designating the sign. 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 increasing in significance from 0 through 7. The value of the unsigned integer is in the range 0 through 255.

- **word**

A word is two contiguous bytes starting on an arbitrary byte boundary or located in a register \( R_n <15:0> \). The bits are numbered from the right 0 through 15. Figure 4-3 illustrates the word format.

![Figure 4-3 Word Format](image)

A word is specified by its address \( A \), the address of the byte containing bit 0. When interpreted as a signed quantity, a word is a 2's complement integer with bits increasing in significance from 0 through 14, and with bit 15 designating the sign. 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 increasing in significance from 0 through 15. The value of the unsigned integer is in the range 0 through 65,535.
Data Representation

- longword

A longword is four contiguous bytes starting on an arbitrary byte boundary or located in a register Rn<31:00>. The bits are numbered from the right 0 through 31. Longword format is illustrated in Figure 4-4.

Figure 4-4  Longword Format

A longword is specified by its address A, the address of the byte containing bit 0. When interpreted as a signed quantity, a longword is a 2's complement integer with bits increasing in significance from 0 through 30, and with bit 31 designating the sign. 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 increasing in significance from 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 increase in significance 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.

- quadword

A quadword is eight contiguous bytes starting on an arbitrary byte boundary or in two consecutive registers R[n+1]’R[n]. The bits are numbered from the right 0 through 63. Quadword format is illustrated in Figure 4-5.

Figure 4-5  Quadword Format
A quadword is specified by its address A, the address of the byte containing bit 0. When interpreted as a signed quantity, a quadword is a 2's complement integer with bits increasing in significance from 0 through 62, and with bit 63 designating the sign. The value of the integer is in the range $-2^{63}$ to $2^{63} - 1$.

**Floating Point Data**
The floating point data types are used to represent approximations to quantities using a scientific notation consisting of a sign, the exponent of a power of two, and a fraction between .5 (inclusive) and 1.0 (exclusive). The value of a floating point number is the sign applied to the fractional part multiplied by two raised to the power specified by the exponent part. VAX-11 supports, in the instruction set, floating point data types of 32 and 64 bits sizes. These are termed single precision floating or simply floating (F
-floating) and double floating (D
-floating) respectively.

- F
-floating (single precision floating)

An F
-floating datum is four contiguous bytes starting on an arbitrary byte boundary or in a register Rn. The bits are labelled from the right 0 through 31. Figure 4-6 illustrates the F
-floating format.

```
<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>7</th>
<th>6</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>EXP</td>
<td>FRACTION</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>: A</td>
<td></td>
<td></td>
</tr>
<tr>
<td>31</td>
<td>16</td>
<td>FRACTION</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>: A+2</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

**Figure 4-6  F
-floating Format**

An F
-floating datum is specified by its address A, the address of the byte containing bit 0. The form of an F
-floating datum is sign magnitude with bit 15 the sign bit, bits 14:7 an excess 128 binary exponent, and bits 6:0 and 31:16 a normalized 24-bit fraction with the redundant most significant fraction bit not represented. Within the fraction, bits increase in significance from 16 through 31 and 0 through 6. The 8-bit exponent field encodes the values 0 through 255. An exponent value of 0 together with a sign bit of 0, is taken to indicate that the F
-floating datum has a value of 0. Exponent values of 1 through 255 indicates true binary exponents of $-127$ through $+127$. An exponent value of 0, together with a sign bit of 1, is taken as reserved. Floating point instructions processing a reserved operand take a reserved operand fault. The magnitude of an F
-floating datum is in the approximate range $0.29 * 10^{-38}$ through $1.7 * 10^{38}$. The precision of an F
-floating datum
is approximately one part in $2^{23}$, i.e., approximately 7 decimal digits.

- D_floating (double precision floating)

The D_floating datum is eight contiguous bytes starting on an arbitrary byte boundary or in two consecutive registers $R[n+1]$*$R[n]$. The bits are labelled from the right 0 through 63. D_floating format is illustrated in Figure 4-7.

![Figure 4-7 D_floating Format](image)

A D_floating datum is specified by its address $A$, the address of the byte containing bit 0. The form of a D_floating datum is identical to the F_floating datum except for an additional 32 low significance fraction bits. Within the fraction, bits increase in significance from 48 through 63, 32 through 47, 16 through 31, and 0 through 6. The exponent conventions, and approximate range of values is the same for both D_floating and F_floating. The precision of a D_floating datum is approximately one part in $2^{55}$, i.e., approximately 16 decimal digits.

**CHARACTER STRING DATA TYPE**

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, searching, concatenating, and translating strings.

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. The format of a character string is illustrated in Figure 4-8.
The address of a string specifies the first character of a string. In the following example, "XYZ" is represented as:

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.

**NUMERIC STRING DATA TYPES**

The numeric 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 presentation is frequently more efficient. The decimal form also provides greater precision than floating point and greater range than integer. There are two forms of decimal data on VAX-11; the decimal string data types in which each decimal digit occupies one byte and a more compact form (discussed in the next section) in which two decimal digits are packed into one byte. These are termed numeric and packed decimal strings respectively. Because the numeric string form must represent many external data arrangements exactly, it appears in several forms. The most significant distinguishing characteristic is whether the sign, if any, appears before the first digit or whether it is superimposed on the final digit. These are termed leading separate and trailing numeric strings respectively.
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 ASCII decimal digit characters (0-9). The representation for the high order digits is:

<table>
<thead>
<tr>
<th>digit</th>
<th>decimal</th>
<th>hex</th>
<th>ASCII character</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>48</td>
<td>30</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>49</td>
<td>31</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>50</td>
<td>32</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>51</td>
<td>33</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>52</td>
<td>34</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>53</td>
<td>35</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>54</td>
<td>36</td>
<td>6</td>
</tr>
<tr>
<td>7</td>
<td>55</td>
<td>37</td>
<td>7</td>
</tr>
<tr>
<td>8</td>
<td>56</td>
<td>38</td>
<td>8</td>
</tr>
<tr>
<td>9</td>
<td>57</td>
<td>39</td>
<td>9</td>
</tr>
</tbody>
</table>

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 normal form is generated on output of all operations. The valid representations of the digit and sign in each of the latter two formats is shown in Table 4-1.
### Table 4-1  Representation of Least Significant Digit and Sign

<table>
<thead>
<tr>
<th>Zoned Numeric Format</th>
<th>ASCII char.</th>
<th>Overpunch Format</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>decimal</td>
<td>hex</td>
</tr>
<tr>
<td>0</td>
<td>48</td>
<td>30</td>
</tr>
<tr>
<td>1</td>
<td>49</td>
<td>31</td>
</tr>
<tr>
<td>2</td>
<td>50</td>
<td>32</td>
</tr>
<tr>
<td>3</td>
<td>51</td>
<td>33</td>
</tr>
<tr>
<td>4</td>
<td>52</td>
<td>34</td>
</tr>
<tr>
<td>5</td>
<td>53</td>
<td>35</td>
</tr>
<tr>
<td>6</td>
<td>54</td>
<td>36</td>
</tr>
<tr>
<td>7</td>
<td>55</td>
<td>37</td>
</tr>
<tr>
<td>8</td>
<td>56</td>
<td>38</td>
</tr>
<tr>
<td>9</td>
<td>57</td>
<td>39</td>
</tr>
<tr>
<td>-0</td>
<td>112</td>
<td>70</td>
</tr>
<tr>
<td>-1</td>
<td>113</td>
<td>71</td>
</tr>
<tr>
<td>-2</td>
<td>114</td>
<td>72</td>
</tr>
<tr>
<td>-3</td>
<td>115</td>
<td>73</td>
</tr>
<tr>
<td>-4</td>
<td>116</td>
<td>74</td>
</tr>
<tr>
<td>-5</td>
<td>117</td>
<td>75</td>
</tr>
<tr>
<td>-6</td>
<td>118</td>
<td>76</td>
</tr>
<tr>
<td>-7</td>
<td>119</td>
<td>77</td>
</tr>
<tr>
<td>-8</td>
<td>120</td>
<td>78</td>
</tr>
<tr>
<td>-9</td>
<td>121</td>
<td>79</td>
</tr>
</tbody>
</table>

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. The following examples illustrate the representations of "123" and "−123" in trailing numeric string format.
Thus "123" is represented as:

**ZONED FORMAT OR UNSIGNED**

<table>
<thead>
<tr>
<th>7</th>
<th>4</th>
<th>3</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>1</td>
<td></td>
<td>: A</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td></td>
<td>: A+1</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td></td>
<td>: A+2</td>
</tr>
</tbody>
</table>

**OVERPUNCH FORMAT**

<table>
<thead>
<tr>
<th>7</th>
<th>4</th>
<th>3</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>1</td>
<td></td>
<td>: A</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td></td>
<td>: A+1</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td></td>
<td>: A+2</td>
</tr>
</tbody>
</table>

and "−123" is represented as:

**ZONED FORMAT**

<table>
<thead>
<tr>
<th>7</th>
<th>4</th>
<th>3</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>1</td>
<td></td>
<td>: A</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td></td>
<td>: A+1</td>
</tr>
<tr>
<td>7</td>
<td>3</td>
<td></td>
<td>: A+2</td>
</tr>
</tbody>
</table>

**OVERPUNCH FORMAT**

<table>
<thead>
<tr>
<th>7</th>
<th>4</th>
<th>3</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>1</td>
<td></td>
<td>: A</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td></td>
<td>: A+1</td>
</tr>
<tr>
<td>4</td>
<td>C</td>
<td></td>
<td>: A+2</td>
</tr>
</tbody>
</table>
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:

<table>
<thead>
<tr>
<th>sign</th>
<th>decimal</th>
<th>hex</th>
<th>ASCII character</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>43</td>
<td>2B</td>
<td>+</td>
</tr>
<tr>
<td>+</td>
<td>32</td>
<td>20</td>
<td>&lt;blank&gt;</td>
</tr>
<tr>
<td>-</td>
<td>45</td>
<td>2D</td>
<td>−</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>digit</th>
<th>decimal</th>
<th>hex</th>
<th>ASCII character</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>48</td>
<td>30</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>49</td>
<td>31</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>50</td>
<td>32</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>51</td>
<td>33</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>52</td>
<td>34</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>53</td>
<td>35</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>54</td>
<td>36</td>
<td>6</td>
</tr>
<tr>
<td>7</td>
<td>55</td>
<td>37</td>
<td>7</td>
</tr>
<tr>
<td>8</td>
<td>56</td>
<td>38</td>
<td>8</td>
</tr>
<tr>
<td>9</td>
<td>57</td>
<td>39</td>
<td>9</td>
</tr>
</tbody>
</table>

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. The following examples illustrate the representations of “+123” and “−123” in leading separate numeric string format.
Thus "+123" is represented as:

<table>
<thead>
<tr>
<th>7</th>
<th>4</th>
<th>3</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td></td>
<td>B</td>
<td>: A</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>1</td>
<td>: A+1</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>2</td>
<td>: A+2</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>3</td>
<td>: A+3</td>
</tr>
</tbody>
</table>

and "−123" is represented as:

<table>
<thead>
<tr>
<th>7</th>
<th>4</th>
<th>3</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td></td>
<td>D</td>
<td>: A</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>1</td>
<td>: A+1</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>2</td>
<td>: A+2</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>3</td>
<td>: A+3</td>
</tr>
</tbody>
</table>

**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:

<table>
<thead>
<tr>
<th>digit or sign</th>
<th>decimal</th>
<th>hex</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
</tbody>
</table>
Data Representation

<table>
<thead>
<tr>
<th>digit or sign</th>
<th>decimal</th>
<th>hex</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>7</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>8</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>9</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>+</td>
<td>10, 12, 14 or 15</td>
<td>A, C, E, or F</td>
</tr>
<tr>
<td>-</td>
<td>11 or 13</td>
<td>B, or D</td>
</tr>
</tbody>
</table>

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. In the following example, "+123" (length 3) is represented in packed decimal format as:

```
<table>
<thead>
<tr>
<th>7</th>
<th>4</th>
<th>3</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>12</td>
<td></td>
</tr>
</tbody>
</table>
```

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

```
<table>
<thead>
<tr>
<th>7</th>
<th>4</th>
<th>3</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>13</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

69
**QUEUE DATA TYPES**

The queue data types consist of absolute queues and self-relative queues. Each type of queue has a matching set of instructions described in Chapter 7. The self-relative queues are intended for use in situations where they are addressed by two separate processes which may view the queues as being two separate locations in their respective virtual address spaces.

The instructions which operate on self-relative queues are interlocked so that as long as only interlocked instructions are used on the queue, the processes may be in separate processors, each directly addressing the queue.

Absolute queues are somewhat simpler in structure than self-relative queues in that their pointers are virtual addresses. Also, the instructions which operate on these queues are not interlocked. Hence, operations on absolute queues are, in general, somewhat faster. However, absolute queues cannot be used when more than one processor can access them. Also, they can only be shared by two processes in the same processor when both processes address the queue in the same section of their virtual address space. Figure 4-9a illustrates the format of the self-relative queue and Figure 4-9b illustrates the format of the absolute queue.
Data Representation

SELF-RELATIVE QUEUE WITH TWO ENTRIES

Figure 4-9a Self-Relative Queues

EMPTY ABSOLUTE QUEUE (HEADER ONLY i.e. SIMPLE ENTRY ONLY)

ABSOLUTE QUEUE WITH HEADER AND OTHER ENTRY

71
**VARIABLE LENGTH BIT FIELD DATA TYPE**

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

**Variable Length Bit Field**

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

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

Figure 4-10a illustrates the variable length bit field where the field is the shaded area.
Data Representation

Figure 4-10a  Variable Length Bit Field

The position $P$ (in bits) can be either a positive or negative displacement within the range $-2^{31}$ through $2^{31}-1$. The position can be viewed as a signed 29-bit byte offset and a 3-bit bit-within-byte field. The position $P$ is illustrated in figure 4-10b.

Figure 4-10b  Bit Position $P$

The sign extended 29-bit byte offset is added to the address $A$ and the resulting address specifies the byte in which the field begins. The 3-bit bit-within-byte field encodes the starting position (0 through 7) of the field within that byte. 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 the 2's complement with bits increasing in significance from 0 through $s-2$, where bit $s-1$ is designated the sign bit. When interpreted as an unsigned integer, bits increase in significance from 0 through $s-1$. 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.

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

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

73
A variable bit field may be contained in zero to five bytes. From a memory management point of view only the minimum number of bytes necessary to contain the field is actually referenced.

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

Example:
The variable length bit field attributes are specified as follows:

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

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

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

The next example illustrates the variable length bit field F with a negative displacement from the byte address A.
**Example:**
The variable length bit field attributes are specified as follows:

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

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

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

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

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

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

The VAX-11 string instructions are unable to process string data types stored in registers. Thus, there is no architectural specification of the representation of strings in registers.
CHAPTER 5

INSTRUCTION FORMATS AND ADDRESSING MODES

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.

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 content of the register is 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 season, 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.

**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.

**Assembler Radix 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

\[ \text{MOVW } \uparrow X \; 3456, \; -(\text{SP}). \]

Examples of hexadecimal numbers and conversion between hexadecimal and decimal are provided in Appendix A.
A. MOVE LONG INSTRUCTION

```
MOV L 6(R1), R5
```

; SIX IS ADDED TO R1, THE RESULT USED AS AN
; ADDRESS AND THE CONTENTS OF THAT ADDRESS
; IS MOVED TO R5

<table>
<thead>
<tr>
<th>BYTE</th>
<th>MOV L</th>
<th>OPCODE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>(R1)</td>
<td>OPERAND SPECIFIER 1</td>
</tr>
<tr>
<td>3</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>R5</td>
<td>OPERAND SPECIFIER 2</td>
</tr>
</tbody>
</table>

B. MOVE WORD INSTRUCTION

```
MOV W #\$X3456, -(SP)
```

; THE NUMBER 3456 IS PUSHED ON THE
; STACK

<table>
<thead>
<tr>
<th>BYTE</th>
<th>MOV W</th>
<th>OPCODE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>(PC)</td>
<td>OPERAND SPECIFIER 1</td>
</tr>
<tr>
<td>3</td>
<td>56</td>
<td>IMMEDIATE DATA (56 STORED IN BYTE 3)</td>
</tr>
<tr>
<td>4</td>
<td>34</td>
<td>(34 STORED IN BYTE 4)</td>
</tr>
<tr>
<td>5</td>
<td>-(SP)</td>
<td>OPERAND SPECIFIER 2</td>
</tr>
</tbody>
</table>

Figure 5-1  General VAX-11 Instruction Format
Instruction Formats and Addressing Modes

C. ADD LONG INSTRUCTION (3 OPERAND)
ADDL 3, (SP) +, R4, R5 ; NUMBER ON THE STACK IS
; ADDED TO THE CONTENTS OF
; R4 AND RESULT IS STORED
; IN R5

<table>
<thead>
<tr>
<th>BYTE</th>
<th>ADDL 3</th>
<th>ADDL 3</th>
<th>(SP) +</th>
<th>OPERAND SPECIFIER 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>ADDL 3</td>
<td>OPERAND SPECIFIER 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>(SP) +</td>
<td>OPERAND SPECIFIER 2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>R4</td>
<td>OPERAND SPECIFIER 3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>R5</td>
<td>OPCODE</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 5-2  Examples of Instruction Format

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.

1 BYTE OPCODE

<table>
<thead>
<tr>
<th>7</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>OPCODE</td>
</tr>
</tbody>
</table>

15 8 7 0

FC-FF
(1111 1100 -1111 1111)

2 BYTE OPCODE

Figure 5-3  Opcode Format

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
- F floating—32-bit single-precision floating point (same as longword for addressing mode considerations)
Instruction Formats and Addressing Modes

- Quadword—64-bit
- D_floating—64-bit double-precision floating point (same as quadword 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 write-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 specifies 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.

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 addressing mode. Depending on the mode, this field is 2, 4, or 8 bits. Most addressing modes include additional information. Depending on the mode, up to two register designators are included.

ADDRESSING MODE

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. This section describes both general mode and 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
**Instruction Formats and Addressing Modes**

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). “Src” designates source. “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.

### Table 5-1 Summary of Addressing Modes

#### GENERAL REGISTER ADDRESSING

<table>
<thead>
<tr>
<th>Hex</th>
<th>Dec</th>
<th>Name</th>
<th>Assembler</th>
<th>r</th>
<th>m</th>
<th>w</th>
<th>a</th>
<th>v</th>
<th>PC</th>
<th>SP</th>
<th>Indexable?</th>
</tr>
</thead>
<tbody>
<tr>
<td>0:3</td>
<td>0:3</td>
<td>literal</td>
<td>S # literal</td>
<td>f</td>
<td>f</td>
<td>f</td>
<td>f</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>indexed</td>
<td>i [Rx]</td>
<td>y</td>
<td>y</td>
<td>y</td>
<td>y</td>
<td>f</td>
<td>y</td>
<td>f</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>register</td>
<td>Rn</td>
<td>y</td>
<td>y</td>
<td>y</td>
<td>y</td>
<td>u</td>
<td>q</td>
<td>f</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>6</td>
<td>register deferred</td>
<td>(Rn)</td>
<td>y</td>
<td>y</td>
<td>y</td>
<td>y</td>
<td>u</td>
<td>y</td>
<td>y</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>7</td>
<td>autodecrement</td>
<td>-(Rn)</td>
<td>y</td>
<td>y</td>
<td>y</td>
<td>y</td>
<td>u</td>
<td>y</td>
<td>u</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>8</td>
<td>autoincrement</td>
<td>(Rn)+</td>
<td>y</td>
<td>y</td>
<td>y</td>
<td>y</td>
<td>p</td>
<td>y</td>
<td>u</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>9</td>
<td>autoincrement</td>
<td>@ (R)+</td>
<td>y</td>
<td>y</td>
<td>y</td>
<td>y</td>
<td>p</td>
<td>y</td>
<td>u</td>
<td></td>
</tr>
<tr>
<td>A</td>
<td>10</td>
<td>byte displacement</td>
<td>B' (Rn)</td>
<td>y</td>
<td>y</td>
<td>y</td>
<td>y</td>
<td>p</td>
<td>y</td>
<td>y</td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>11</td>
<td>byte displacement</td>
<td>D (Rn)</td>
<td>y</td>
<td>y</td>
<td>y</td>
<td>y</td>
<td>p</td>
<td>y</td>
<td>y</td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>12</td>
<td>word displacement</td>
<td>D (Rn)</td>
<td>y</td>
<td>y</td>
<td>y</td>
<td>y</td>
<td>p</td>
<td>y</td>
<td>y</td>
<td></td>
</tr>
<tr>
<td>D</td>
<td>13</td>
<td>word displacement</td>
<td>D (Rn)</td>
<td>y</td>
<td>y</td>
<td>y</td>
<td>y</td>
<td>p</td>
<td>y</td>
<td>y</td>
<td></td>
</tr>
<tr>
<td>E</td>
<td>14</td>
<td>longword displacement</td>
<td>D (Rn)</td>
<td>y</td>
<td>y</td>
<td>y</td>
<td>y</td>
<td>p</td>
<td>y</td>
<td>y</td>
<td></td>
</tr>
<tr>
<td>F</td>
<td>15</td>
<td>longword displacement</td>
<td>D (Rn)</td>
<td>y</td>
<td>y</td>
<td>y</td>
<td>y</td>
<td>p</td>
<td>y</td>
<td>y</td>
<td></td>
</tr>
</tbody>
</table>

#### PROGRAM COUNTER ADDRESSING

<table>
<thead>
<tr>
<th>Hex</th>
<th>Dec</th>
<th>Name</th>
<th>Assembler</th>
<th>r</th>
<th>m</th>
<th>w</th>
<th>a</th>
<th>v</th>
<th>PC</th>
<th>SP</th>
<th>Indexable?</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>8</td>
<td>immediate</td>
<td>l # constant</td>
<td>u</td>
<td>u</td>
<td>u</td>
<td>u</td>
<td>y</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>9</td>
<td>absolute</td>
<td>@ # address</td>
<td>y</td>
<td>y</td>
<td>y</td>
<td>y</td>
<td>y</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A</td>
<td>10</td>
<td>byte relative</td>
<td>B # address</td>
<td>y</td>
<td>y</td>
<td>y</td>
<td>y</td>
<td>y</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>11</td>
<td>byte relative</td>
<td>B # address</td>
<td>y</td>
<td>y</td>
<td>y</td>
<td>y</td>
<td>y</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>12</td>
<td>word relative</td>
<td>W # address</td>
<td>y</td>
<td>y</td>
<td>y</td>
<td>y</td>
<td>y</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>D</td>
<td>13</td>
<td>word relative</td>
<td>W # address</td>
<td>y</td>
<td>y</td>
<td>y</td>
<td>y</td>
<td>y</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>E</td>
<td>14</td>
<td>longword relative</td>
<td>L # address</td>
<td>y</td>
<td>y</td>
<td>y</td>
<td>y</td>
<td>y</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>F</td>
<td>15</td>
<td>longword relative</td>
<td>L # address</td>
<td>y</td>
<td>y</td>
<td>y</td>
<td>y</td>
<td>y</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **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

84
GENERAL MODE ADDRESSING
Register Mode

Assembler
Syntax: Rn
Mode
Specifier: 5
Operand
Specifier
Format:

\[
\begin{array}{cccc}
7 & 4 & 3 & 0 \\
 & & 5 & \text{Rn} \\
\end{array}
\]

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 pro-
vide speed advantages when used for operating on
frequently-accessed variables.

Special
Comments: This mode can be used with operand specifiers us-
ing 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
that takes two registers, the content 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 INSTRUCCION

Instruction Format: MOVW R1, R2

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

BEFORE INSTRUCTION EXECUTION

\[
\begin{array}{cccccc}
0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & 2 & 3 \\
C & O & A & 0 & 3 & 4 \\
\end{array}
\]

AFTER INSTRUCTION EXECUTION

\[
\begin{array}{cccccc}
0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 3 & 4 & 1 \\
C & O & A & 0 & 3 & 4 \\
\end{array}
\]

MACHINE CODE: ASSUME STARTING LOCATION 00003000

\[
\begin{array}{c}
00003000 \quad 00003001 \quad 00003002 \\
\hline
B0 \quad 51 \quad 52 \\
\end{array}
\]

OPCODE FOR MOVE WORD INSTRUCTION
OPERAND SPECIFIER, SOURCE; REGISTER MODE 1
OPERAND SPECIFIER, DESTINATION; REGISTER MODE 2

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

Register Deferred Mode

Assembler Syntax: (Rn)

Mode Specifier: 6

Operand Specifier Format:
**Instruction Formats and Addressing Modes**

**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**

<table>
<thead>
<tr>
<th>ADDRESS SPACE</th>
<th>00001010</th>
<th>00001011</th>
<th>00001012</th>
<th>00001013</th>
<th>00001014</th>
<th>00001015</th>
<th>00001016</th>
<th>00001017</th>
</tr>
</thead>
<tbody>
<tr>
<td>00001010</td>
<td>AB</td>
<td>CD</td>
<td>EF</td>
<td>12</td>
<td>34</td>
<td>56</td>
<td>76</td>
<td>65</td>
</tr>
</tbody>
</table>

**AFTER INSTRUCTION EXECUTION**

<table>
<thead>
<tr>
<th>ADDRESS SPACE</th>
<th>00001010</th>
<th>00001011</th>
<th>00001012</th>
<th>00001013</th>
<th>00001014</th>
<th>00001015</th>
<th>00001016</th>
<th>00001017</th>
</tr>
</thead>
<tbody>
<tr>
<td>00001010</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
</tr>
</tbody>
</table>
Instruction Formats and Addressing Modes

MACHINE CODE: ASSUME STARTING LOCATION 00003000

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

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

Autoincrement Mode

Assembler Syntax: (Rn)+

Mode Specifier: 8

OperandSpecifier Format:

<table>
<thead>
<tr>
<th>7</th>
<th>4</th>
<th>3</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>Rn</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

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

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

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

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

**MACHINE CODE:** ASSUME STARTING LOCATION 3000

<table>
<thead>
<tr>
<th>Address</th>
<th>Opcode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>00003000</td>
<td>D0</td>
<td>OPCODE FOR MOVE LONG WORD INSTRUCTION</td>
</tr>
<tr>
<td>00003001</td>
<td>81</td>
<td>AUTOINCREMENT MODE, REGISTER R1</td>
</tr>
<tr>
<td>00003002</td>
<td>52</td>
<td>REGISTER MODE, REGISTER R2</td>
</tr>
</tbody>
</table>
This example shows a Move Long instruction using autoincrement mode. The content of register R1 is the effective address of the source operand. Since the operand is a 32-bit longword, four bytes are transferred to register R2. R1 is then incremented by 4 since the instruction specifies a longword data type.

**Autoincrement Deferred Mode**

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

**Mode Specifier:** 9

**Operand Specifier Format:**

<table>
<thead>
<tr>
<th>7</th>
<th>4</th>
<th>3</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Rn</td>
<td></td>
</tr>
</tbody>
</table>

**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 contents of register Rn and the contents of register Rn are 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 (refer to absolute mode).

**EXAMPLE:** AUTOINCREMENT DEFERRED MODE, MOVE WORD INSTRUCTION
Instruction Formats and Addressing Modes

Instruction Format: MOVW @(R1)+, R2

BEFORE INSTRUCTION EXECUTION

ADDRESS
SPACE

00001010 00
00001011 11
00001012 22
00001013 33
00001014 44
00001015 55

OPERAND ADDRESS
33221100

33221100 34
33221101 5F
33221102 00
33221103 00

AFTER INSTRUCTION EXECUTION

R1
00001014
R2
00005F34

MACHINE CODE: ASSUME STARTING LOCATION 00003000

00003000 80
00003001 91
00003002 52

OPCODE FOR MOVE WORD INSTRUCTION
AUTOINCREMENT DEFERRED MODE, REGISTER R1
REGISTER MODE, REGISTER R2

This example shows a Move Word instruction using autoincrement deferred mode. The content 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; the upper half of R2 unaltered. R1 is then incremented by 4 since it points to a 32-bit address.
Autodecrement Mode

Assembler Syntax:  

\[-(Rn)\]

Mode Specifier:  

7

Operand Specifier Format:

\[
\begin{array}{c}
7 \\
4 \\
3 \\
Rn \\
0
\end{array}
\]

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 content of register Rn is the address of the operand.

Special Comments:  

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

EXAMPLE:  

AUTODECREMENT MODE, MOVE LONG INSTRUCTION MOVL -(R3), R4

BEFORE INSTRUCTION EXECUTION

\[
\begin{array}{c}
00001014 \\
00001015 \\
00001016 \\
00001017 \\
R3 \\
R4
\end{array}
\]

AFTER INSTRUCTION EXECUTION

\[
\begin{array}{c}
00001014 \\
CE543210 \\
00001018 \\
00000000 \\
R3 \\
R4
\end{array}
\]
This example shows a Move Long instruction using autodecrement mode. The contents of register R3 are 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 are then used as the address of the operand. The instruction causes the operand to be fetched and loaded into register R4.

**Literal Mode**

**Assembler Syntax:**

S↑# literal

**Mode Specifier:**

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

**OperandSpecifier Format:**

```
7 6 5 0
0 0 LITERAL
```

**Description:**

Literal mode addressing provides an efficient means of specifying integer constants in the range from 0 to 63 (decimal). This is called short literal. Literal values above 63 can be obtained by immediate mode (autoincrement mode using the PC) although immediate mode is longer. For predefined values the assembler will choose between short literal and immediate modes. For short literal operands, the format is:
Bits 7 and 6, however, are always set to zero. The following examples show some short literals; the literals are 14, 30, 46, and 62.

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

![Diagram of EXP and FRAC fields]

**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 the decimal literal of 12 to be represented by a fraction of 4 and an exponent of 4.

95
Instruction Formats and Addressing Modes

**LITERAL MODE**

```
  7 6 5 4 3 2 1 0
  0 0 1 0 0 1 0 0
```

```
15 14 13 12 11 10 9 8 7 6 5 4 3 0
0 1 0 0 0 1 0 0 1 0 0 0
```

**FLOATING OPERAND**

---

**Table 5-2  Floating Literals**

<table>
<thead>
<tr>
<th>Exponent</th>
<th>FRACTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1/2</td>
</tr>
<tr>
<td>1</td>
<td>1 1/8</td>
</tr>
<tr>
<td>2</td>
<td>2 1/4</td>
</tr>
<tr>
<td>3</td>
<td>4 1/2</td>
</tr>
<tr>
<td>4</td>
<td>8</td>
</tr>
<tr>
<td>5</td>
<td>16</td>
</tr>
<tr>
<td>6</td>
<td>32</td>
</tr>
<tr>
<td>7</td>
<td>64</td>
</tr>
</tbody>
</table>

| 1         | 9/16     |
| 2         | 1 1/8    |
| 3         | 2 1/4    |
| 4         | 4 1/2    |
| 5         | 8        |
| 6         | 16       |
| 7         | 32       |

| 2         | 5/8      |
| 3         | 1 3/8    |
| 4         | 2 3/4    |
| 5         | 5 1/2    |
| 6         | 11       |
| 7         | 22       |

| 3         | 3/4      |
| 4         | 6        |
| 5         | 12       |
| 6         | 24       |
| 7         | 32       |

| 4         | 13/16    |
| 5         | 1 5/8    |
| 6         | 3 3/4    |
| 7         | 6 1/2    |

| 5         | 7        |
| 6         | 13       |
| 7         | 26       |

| 6         | 14       |
| 7         | 30       |

| 7         | 15       |
| 6         | 30       |
| 7         | 40       |
| 8         | 80       |

| 8         | 104      |
| 9         | 120      |

---

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

MOVL $9, R4

**BEFORE INSTRUCTION EXECUTION**

```
R4
000000000
```

**AFTER INSTRUCTION EXECUTION**

```
R4
000000009
```
Instruction Formats and Addressing Modes

MACHINE CODE: ASSUME STARTING LOCATION 00003000

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

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

Displacement Mode

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

Mode Specifier: A—(byte displacement)
C—(word displacement)
E—(longword displacement)

Operand Specifier Format:

<table>
<thead>
<tr>
<th>15</th>
<th>8</th>
<th>7</th>
<th>4</th>
<th>3</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>DISP.</td>
<td>A</td>
<td>Rn</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>23</th>
<th>8</th>
<th>7</th>
<th>4</th>
<th>3</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>DISP.</td>
<td>C</td>
<td>Rn</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>39</th>
<th>8</th>
<th>7</th>
<th>4</th>
<th>3</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>DISP.</td>
<td>E</td>
<td>Rn</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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.
Instruction Formats and Addressing Modes

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 (refer to relative mode).

EXAMPLE:

DISPLACEMENT MODE, MOVE BYTE INSTRUCTION
MOV B↑5(R4), B↑3(R3)

BEFORE INSTRUCTION EXECUTION

<table>
<thead>
<tr>
<th>ADDRESS SPACE</th>
<th>R4</th>
<th>R3</th>
</tr>
</thead>
<tbody>
<tr>
<td>00001015</td>
<td>00</td>
<td>000001012</td>
</tr>
<tr>
<td>00001016</td>
<td>00</td>
<td>00001012</td>
</tr>
<tr>
<td>00001017</td>
<td>06</td>
<td>000001017</td>
</tr>
<tr>
<td>00001018</td>
<td>00</td>
<td>00010</td>
</tr>
<tr>
<td>00001019</td>
<td>00</td>
<td>00001018</td>
</tr>
<tr>
<td>00002021</td>
<td>00</td>
<td></td>
</tr>
<tr>
<td>00002022</td>
<td>00</td>
<td></td>
</tr>
<tr>
<td>00002023</td>
<td>00</td>
<td></td>
</tr>
</tbody>
</table>

AFTER INSTRUCTION EXECUTION

<table>
<thead>
<tr>
<th>ADDRESS SPACE</th>
<th>R4</th>
<th>R3</th>
</tr>
</thead>
<tbody>
<tr>
<td>00001015</td>
<td>00</td>
<td>000001012</td>
</tr>
<tr>
<td>00001016</td>
<td>00</td>
<td>00001012</td>
</tr>
<tr>
<td>00001017</td>
<td>06</td>
<td>00002000</td>
</tr>
<tr>
<td>00001018</td>
<td>00</td>
<td>00002000</td>
</tr>
<tr>
<td>00002021</td>
<td>00</td>
<td></td>
</tr>
<tr>
<td>00002022</td>
<td>00</td>
<td></td>
</tr>
<tr>
<td>00002023</td>
<td>06</td>
<td></td>
</tr>
</tbody>
</table>

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 R4 to form the address of the byte operand. The operand is moved to the address formed by adding the displacement of 3 to the contents of R3.

Displacement Deferred Mode

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

Mode
Specifier:  
B—(byte displacement)  
D—(word displacement)  
F—(longword displacement)

Operand
Specifier
Format:

\[
\begin{array}{cccccc}
15 & 8 & 7 & 4 & 3 & 0 \\
\text{DISP.} & B & F & \text{SPECIFIER EXTENSION IS BYTE DISPLACEMENT DEFERRED} \\
23 & 8 & 7 & 4 & 3 & 0 \\
\text{DISP.} & D & F & \text{SPECIFIER EXTENSION IS WORD DISPLACEMENT DEFERRED} \\
39 & 8 & 7 & 4 & 3 & 0 \\
\text{DISP.} & F & F & \text{SPECIFIER EXTENSION IS LONG WORD DISPLACEMENT DEFERRED} \\
\end{array}
\]

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 Rn and the result is a longword address of the operand address.

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

EXAMPLE:  
DISPLACEMENT DEFERRED MODE, INCREMENT WORD INSTRUCTION  
INCW @B↑5(R4)
This example shows an Increment Word instruction using displacement deferred mode. The quantity 5 is added to the contents of R4 to produce the longword address of the address of the operand. The operand of 5713 is incremented to 5714.

**INDEX MODE**

**Assembler Syntax:**

\[ i[R_x] \]

**Mode Specifier:**

4
Instruction Formats and Addressing Modes

Operand Specifier Format:

<table>
<thead>
<tr>
<th>15</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
</tr>
</tbody>
</table>

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 quadword 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 where the number in the index register is shifted left by the context of the data type specified (none for byte, once for word, twice for longword, three times for quadword). 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 indexed register Rx:

1. The PC cannot be used as an index register. If it is, a reserved addressing mode fault occurs.
2. If the base operand specifier is for an address-
Instruction Formats and Addressing Modes

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

### Table 5-3  Index Mode Addressing

<table>
<thead>
<tr>
<th>MODE</th>
<th>ASSEMBLER NOTATION</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register deferred index</td>
<td>(Rn) [Rx]</td>
</tr>
<tr>
<td>Autoincrement indexed</td>
<td>(Rn) + [Rx]</td>
</tr>
<tr>
<td>Immediate indexed</td>
<td># constant [Rx] which is recognized by assembler but is not generally useful. Operand address is independent of value of constant.</td>
</tr>
<tr>
<td>Autodecrement deferred indexed</td>
<td>@(Rn) + [Rx]</td>
</tr>
<tr>
<td>Absolute indexed</td>
<td>@#address [Rx]</td>
</tr>
<tr>
<td>Autodecrement indexed</td>
<td>-(Rn) [Rx]</td>
</tr>
<tr>
<td>Byte, word or longword displacement indexed</td>
<td>B↑D(Rn) [Rx]</td>
</tr>
<tr>
<td></td>
<td>W↑D(Rn) [Rx]</td>
</tr>
<tr>
<td></td>
<td>L↑D(Rn) [Rx]</td>
</tr>
<tr>
<td>Byte, word or longword displacement deferred indexed</td>
<td>@B↑D(Rn) [Rx]</td>
</tr>
<tr>
<td></td>
<td>@W↑D(Rn) [Rx]</td>
</tr>
<tr>
<td></td>
<td>@L↑D(Rn) [Rx]</td>
</tr>
</tbody>
</table>
Instruction Formats and Addressing Modes

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

**EXAMPLE:**

REGISTER DEFERRED INDEXED MODE, INCREMENT WORD INSTRUCTION

INCW (R2) [R5]

**BEFORE INSTRUCTION EXECUTION**

<table>
<thead>
<tr>
<th>ADDRESS SPACE</th>
<th>R2</th>
<th>R5</th>
</tr>
</thead>
<tbody>
<tr>
<td>00001012</td>
<td>04</td>
<td>00001012</td>
</tr>
<tr>
<td>00001013</td>
<td>56</td>
<td>00000003</td>
</tr>
<tr>
<td>00001014</td>
<td>78</td>
<td></td>
</tr>
<tr>
<td>00001015</td>
<td>87</td>
<td></td>
</tr>
<tr>
<td>00001018</td>
<td>45</td>
<td></td>
</tr>
<tr>
<td>00001019</td>
<td>67</td>
<td></td>
</tr>
</tbody>
</table>

3 bytes \(\times\) 2 BYTES PER WORD = 6

00001012 + 6

00001018

**AFTER INSTRUCTION EXECUTION**

<table>
<thead>
<tr>
<th>ADDRESS SPACE</th>
<th>R2</th>
<th>R5</th>
</tr>
</thead>
<tbody>
<tr>
<td>00001018</td>
<td>46</td>
<td>00001012</td>
</tr>
<tr>
<td>00001019</td>
<td>67</td>
<td>00000003</td>
</tr>
</tbody>
</table>

**ASSEMBLY CODE:**

ASSUME STARTING LOCATION 00003000

<table>
<thead>
<tr>
<th>ADDRESS SPACE</th>
<th>OPCODE</th>
<th>INDEX MODE, REGISTER R5</th>
<th>REGISTER DEFERRED MODE, REGISTER R2</th>
</tr>
</thead>
<tbody>
<tr>
<td>00003000</td>
<td>B6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>00003001</td>
<td>45</td>
<td></td>
<td></td>
</tr>
<tr>
<td>00003002</td>
<td>62</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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.
EXAMPLES: AUTOINCREMENT INDEXED MODE, CLEAR LONG-WORD INSTRUCTION
CLRL (R4) + [R5]

BEFORE INSTRUCTION EXECUTION

ADDRESS
SPACE

000010A6
000010A7
000010A8
000010A9

11
22
33
44

OPERAND

R4
00001012
R5
00000025

INDEX
= 25 x 4 BYTES PER
LONGWORD
= 94

ADDRESS OF OPERAND
000010A6

AFTER INSTRUCTION EXECUTION

R4
000010A6
000010A7
000010A8
000010A9

00
00
00
00

R5
00001016
00000025

MACHINE CODE: ASSUME STARTING LOCATION 00003000

00003000
00003001
00003002

D4
45
84

OPCODE FOR CLEAR LONGWORD INSTRUCTION
INDEX MODE, REGISTER R5
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]

BEFORE INSTRUCTION EXECUTION

ADDRESS SPACE

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>00001012</td>
<td>43</td>
</tr>
<tr>
<td>00001013</td>
<td>21</td>
</tr>
<tr>
<td>00001014</td>
<td>08</td>
</tr>
<tr>
<td>00001015</td>
<td>06</td>
</tr>
</tbody>
</table>

OPERAND ADDRESS

\[ 5 \times 2 \text{ bytes per word} = 0000000A \]

ADDRESS SPACE

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>06082143</td>
<td>22</td>
</tr>
<tr>
<td>06082144</td>
<td>33</td>
</tr>
<tr>
<td>06082145</td>
<td>56</td>
</tr>
</tbody>
</table>

OPERAND

AFTER INSTRUCTION EXECUTION

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0608214D</td>
<td>00</td>
</tr>
<tr>
<td>0608214E</td>
<td>00</td>
</tr>
<tr>
<td>0608214F</td>
<td>56</td>
</tr>
</tbody>
</table>

MACHINE CODE: ASSUME STARTING LOCATION 00003000

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>00003000</td>
<td>84</td>
</tr>
<tr>
<td>00003001</td>
<td>45</td>
</tr>
<tr>
<td>00003002</td>
<td>94</td>
</tr>
</tbody>
</table>

OPCODE FOR CLEAR WORD INSTRUCTION
INDEX MODE, REGISTER R5
AUTOINCREMENT DEFERRED MODE, REGISTER R4

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
CLR W#-(R2) [R4]

**BEFORE INSTRUCTION EXECUTION**

<table>
<thead>
<tr>
<th>ADDRESS SPACE</th>
<th>R2</th>
<th>R4</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000101A</td>
<td>33</td>
<td>00001016</td>
</tr>
<tr>
<td>0000101B</td>
<td>33</td>
<td>00000003</td>
</tr>
<tr>
<td>0000101C</td>
<td>33</td>
<td></td>
</tr>
<tr>
<td>0000101D</td>
<td>33</td>
<td></td>
</tr>
</tbody>
</table>

$3 \times 2$ BYTES PER WORD = 6 (INDEX)

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>00001016</td>
<td>DECREMENT BY 2</td>
</tr>
<tr>
<td>00001014</td>
<td>OPERAND ADDRESS</td>
</tr>
<tr>
<td>00000006</td>
<td>INDEX VALUE</td>
</tr>
<tr>
<td>0000101A</td>
<td>INDEXED OPERAND ADDRESS</td>
</tr>
</tbody>
</table>

**AFTER INSTRUCTION EXECUTION**

<table>
<thead>
<tr>
<th>ADDRESS SPACE</th>
<th>R2</th>
<th>R4</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000101A</td>
<td>00</td>
<td>00001014</td>
</tr>
<tr>
<td>0000101B</td>
<td>00</td>
<td>00000003</td>
</tr>
<tr>
<td>0000101C</td>
<td>33</td>
<td></td>
</tr>
<tr>
<td>0000101D</td>
<td>33</td>
<td></td>
</tr>
</tbody>
</table>

**MACHINE CODE:** ASSUME STARTING LOCATION 00003000

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>00003000</td>
<td>84</td>
</tr>
<tr>
<td>00003001</td>
<td>44</td>
</tr>
<tr>
<td>00003002</td>
<td>72</td>
</tr>
</tbody>
</table>

OPCODE FOR CLEAR WORD INSTRUCTION
INDEX MODE, REGISTER R4
AUTOINCREMENT MODE, REGISTER R2

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 @ #fX1012 [R2]**

**BEFORE INSTRUCTION EXECUTION**

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1026</td>
<td>45</td>
<td></td>
<td>R2 00000005</td>
</tr>
<tr>
<td>1027</td>
<td>36</td>
<td></td>
<td>(5_{16} \times 4 = 14_{16})</td>
</tr>
<tr>
<td>1028</td>
<td>81</td>
<td></td>
<td>00001012</td>
</tr>
<tr>
<td>1029</td>
<td>43</td>
<td></td>
<td>00000014</td>
</tr>
</tbody>
</table>

**AFTER INSTRUCTION EXECUTION**

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1026</td>
<td>00</td>
<td></td>
<td>R2 00000005</td>
</tr>
<tr>
<td>1027</td>
<td>00</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1028</td>
<td>00</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1029</td>
<td>00</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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, \(5 \times 4 = 14_{16}\), which becomes the index value. This value is added to 00001012 yielding 0001026. This is the operand address and four bytes are cleared since a longword data type has been specified.
EXAMPLE: DISPLACEMENT INDEXED MODE, CLEAR QUAD-WORD INSTRUCTION
CLRQ 2(R1) [R3]

BEFORE INSTRUCTION EXECUTION

ADDRESS SPACE
0000402A 24
0000402B 68
0000402C 13
0000402D 57
0000402E 62
0000402F 43
00004030 34
00004031 47

R1

R3
00004000
00000002
00000005

5₁₆ x 8 BYTES PER QUAD WORD
= 2₈₁₆ (INDEX)

00004000
CONTENTS OF R1
00000002
BYTE DISPLACEMENT

0000402A
OPERAND ADDRESS
00000028
INDEX
00004002
INDEXED OPERAND ADDRESS

AFTER INSTRUCTION EXECUTION

ADDRESS SPACE
0000402A 00
0000402B 00
0000402C 00
0000402D 00
0000402E 00
0000402F 00
00004030 00
00004031 00

R1

R3
00004000
00000005

MACHINE CODE: ASSUME STARTING LOCATION 00003000

00003000 7C OPCODE FOR CLEAR QUAD WORD
00003001 43 INDEX MODE, REGISTER R3
00003002 61 REGISTER DEFERRED MODE, REGISTER R1

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
MOVIL @ 1X14 (R1) [R3], R5

BEFORE INSTRUCTION EXECUTION

<table>
<thead>
<tr>
<th>ADDRESS SPACE</th>
<th>R1</th>
<th>R3</th>
<th>R5</th>
</tr>
</thead>
<tbody>
<tr>
<td>00001012</td>
<td>12</td>
<td>00001012</td>
<td></td>
</tr>
<tr>
<td>00001013</td>
<td>34</td>
<td>00000004</td>
<td></td>
</tr>
<tr>
<td>00001014</td>
<td>56</td>
<td>00000000</td>
<td></td>
</tr>
<tr>
<td>00001015</td>
<td>78</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

4₁₆× 4 BYTES PER LONGWORD  = 1₀₁₆ (INDEX)

<table>
<thead>
<tr>
<th></th>
<th>CONTENTS OF R1</th>
<th>DISPLACEMENT</th>
<th>ADDRESS OF OPERAND ADDRESS</th>
</tr>
</thead>
<tbody>
<tr>
<td>00001026</td>
<td>11</td>
<td>00001012</td>
<td>00001026</td>
</tr>
<tr>
<td>00001027</td>
<td>22</td>
<td>00000014</td>
<td></td>
</tr>
<tr>
<td>00001028</td>
<td>33</td>
<td>00001026</td>
<td></td>
</tr>
<tr>
<td>00001029</td>
<td>44</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>OPERAND</th>
<th>44332221</th>
<th>OPERAND ADDRESS</th>
</tr>
</thead>
<tbody>
<tr>
<td>01</td>
<td>44332211</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>INDEX</th>
<th>44332223</th>
</tr>
</thead>
<tbody>
<tr>
<td>45</td>
<td>44332221</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>OPERAND</th>
<th>44332224</th>
</tr>
</thead>
<tbody>
<tr>
<td>67</td>
<td>44332221</td>
</tr>
</tbody>
</table>

AFTER INSTRUCTION EXECUTION

<table>
<thead>
<tr>
<th>R1</th>
<th>R3</th>
<th>R5</th>
</tr>
</thead>
<tbody>
<tr>
<td>00001012</td>
<td>00000004</td>
<td>67452301</td>
</tr>
</tbody>
</table>

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 quality is added to the index yielding the indexed operand address of 44322221. The contents of this address are then moved into register R5 as shown.
### PROGRAM COUNTER ADDRESSING MODES

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.

<table>
<thead>
<tr>
<th>MODE</th>
<th>NAME</th>
<th>ASSEMBLER</th>
<th>FUNCTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>Immediate</td>
<td>I↑#Operand</td>
<td>Constant operand follows address mode</td>
</tr>
<tr>
<td>9</td>
<td>Absolute</td>
<td>@#Location</td>
<td>Absolute address follows address mode</td>
</tr>
<tr>
<td>A</td>
<td>Byte relative</td>
<td>B↑G (R)</td>
<td>Displacement is added to current value of PC to obtain operand address</td>
</tr>
<tr>
<td>C</td>
<td>Word relative</td>
<td>W↑G (R)</td>
<td></td>
</tr>
<tr>
<td>E</td>
<td>Longword relative</td>
<td>L↑G (R)</td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>Byte relative</td>
<td>@B↑G (R)</td>
<td>Displacement is added to current value of PC to yield address of operand address</td>
</tr>
<tr>
<td>D</td>
<td>Word relative</td>
<td>@W↑G (R)</td>
<td></td>
</tr>
<tr>
<td>F</td>
<td>Longword relative</td>
<td>@L↑G (R)</td>
<td></td>
</tr>
</tbody>
</table>
Immediate Mode — same as autoincrement mode, with PC used as general register.

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

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

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

When a standard program is available for different users, it is often helpful to be able to run it at different areas of virtual memory. 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.

Immediate Mode

Assembler Syntax: \texttt{I\#\texttt{operand}}

Mode Specifier: 8

Operand Specifier Format:

\begin{center}
\begin{tabular}{|c|c|c|c|}
\hline
\text{CONSTANT} & 8 & 3 & 0 \\
\hline
\end{tabular}
\end{center}

Description: The immediate addressing mode is autoincrement mode when the PC is used as the general register. The contents of the location following the addressing mode are immediate data.
EXAMPLE: IMMEDIATE MODE, MOVE LONG INSTRUCTION
MOVL #6, R4

BEFORE INSTRUCTION EXECUTION

PC

00001012 00001013 00001014 00001015 00001016 00001017 00001018
D0 8F 06 00 00 00 54

OPCODE FOR MOVE LONG INSTRUCTION
OPERAND SPECIFIER, AUTOINCREMENT PC (IMMEDIATE)

IMMEDIATE DATA

REGISTER MODE, REGISTER R4

00000000

AFTER INSTRUCTION EXECUTION

00001012 00001013 00001014 00001015 00001016 00001017
D0 8F 06 00 00 00

IMMEDIATE DATA

R4

00000006

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.

Absolute Mode

Assembler Syntax: @#location

Mode Specifier: 9

112
Operand Specifier Format:

<table>
<thead>
<tr>
<th>39</th>
<th>8</th>
<th>7</th>
<th>4</th>
<th>3</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADDRESS</td>
<td>9</td>
<td>F</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

**EXAMPLE:** Absolute Mode, Clear Long Instruction

**CLRL @#X674533**

*Before Instruction Execution*

<table>
<thead>
<tr>
<th>PC</th>
<th>ADDRESS SPACE</th>
</tr>
</thead>
<tbody>
<tr>
<td>00001012</td>
<td>D4</td>
</tr>
<tr>
<td>00001013</td>
<td>9F</td>
</tr>
<tr>
<td>00001014</td>
<td>33</td>
</tr>
<tr>
<td>00001015</td>
<td>45</td>
</tr>
<tr>
<td>00001016</td>
<td>67</td>
</tr>
<tr>
<td>00001017</td>
<td>00</td>
</tr>
<tr>
<td>00001018</td>
<td>55</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ADDRESS</th>
<th>OPERAND ADDRESS</th>
</tr>
</thead>
<tbody>
<tr>
<td>00674533</td>
<td>23</td>
</tr>
<tr>
<td>00674534</td>
<td>45</td>
</tr>
<tr>
<td>00674535</td>
<td>72</td>
</tr>
<tr>
<td>00674536</td>
<td>83</td>
</tr>
</tbody>
</table>

*After Instruction Execution*

| 00674533 | 00 |
| 00674534 | 00 |
| 00674535 | 00 |
| 00674536 | 00 |

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

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

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

Operand Specifier Format:

![Diagram showing operand specifier formats for byte, word, and longword displacements.]

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

EXAMPLE:  
RELATIVE MODE, MOVE LONGWORD INSTRUCTION  
MOVL ↑X2016, R4
This example shows a Move Long instruction using relative mode. The word following the address mode is added to the PC to obtain the address of the operand.

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

**Relative Deferred Mode**

**Assembler Syntax:**
- @ B↑D—Byte displacement deferred
- @ W↑D—Word displacement deferred
- @ L↑D—Longword displacement deferred

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

BEFORE INSTRUCTION EXECUTION

<table>
<thead>
<tr>
<th>PC</th>
<th>MOVE LONG_OPCODE</th>
<th>BYTE DISPLACEMENT FROM PC</th>
<th>AMOUNT OF DISPLACEMENT</th>
<th>REGISTER_MODE, REGISTER_2</th>
<th>DISPLACEMENT_CALCULATION</th>
</tr>
</thead>
<tbody>
<tr>
<td>00002000</td>
<td>00</td>
<td>00000000</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00002001</td>
<td>60</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00002002</td>
<td>00</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00002003</td>
<td>00</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>OPERAND ADDRESS</th>
</tr>
</thead>
<tbody>
<tr>
<td>00002050</td>
</tr>
<tr>
<td>00002051</td>
</tr>
<tr>
<td>00002052</td>
</tr>
<tr>
<td>00002053</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>OPERAND</th>
</tr>
</thead>
<tbody>
<tr>
<td>00006000</td>
</tr>
<tr>
<td>00006001</td>
</tr>
<tr>
<td>00006002</td>
</tr>
<tr>
<td>00006003</td>
</tr>
</tbody>
</table>

AFTER INSTRUCTION EXECUTION

R2

01234567
This example shows a Move Long instruction where 00002050 represents the address of the operand. A byte displacement would be selected by the assembler since the displacement is within 128 (decimal) addressable bytes. When the displacement is evaluated, the program counter is pointing to 00002003. The displacement of 4D is added to the current value of the PC yielding the address of 00002050. The contents of this address are then used as the effective operand address of 00006000, and the operand of 1234567 is moved to R2.

**BRANCH ADDRESSING**

**Assembler Syntax:** A

**Mode Specifier:** None

**Operand Specifier Format:**

\[
\begin{array}{c}
7 & 0 \\
\hline
\end{array}
\]

\[
\begin{array}{c}
15 & 0 \\
\hline
\end{array}
\]

**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 content of the PC is the address of the first byte beyond the operand specifier.
The assembler notation for byte and word branch displacement addressing is A where A is the branch address. Note that the branch address and not the displacement is used.

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

**EXAMPLE:**

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

CMPB C, #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 is 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 is bit 0 of B is set (= 1)
CHAPTER 6
INTEGER AND FLOATING POINT INSTRUCTIONS

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, there is an ADD instruction for each of the five integer and floating point data types. The instructions are available symmetrically for each of the three integer 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 independent of the data type.

To simplify understanding of the instruction set, the instruction mnemonics are formed by combining an operation prefix with a data type suffix. 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.

<table>
<thead>
<tr>
<th>INSTRUCTION</th>
<th>DATA TYPE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOVE, CLearR, Move NEGative, CoMPare, TeST</td>
<td>Byte, Word, Longword, Floating, Double</td>
</tr>
<tr>
<td>Move Complement, BIt Test</td>
<td>Byte, Word, Longword</td>
</tr>
</tbody>
</table>
### Integer and Floating Point Instructions

<table>
<thead>
<tr>
<th>INSTRUCTION</th>
<th>DATA TYPE</th>
<th>NUMBER OF OPERANDS</th>
</tr>
</thead>
<tbody>
<tr>
<td>ConVerT</td>
<td></td>
<td>Byte, Word, Longword, Floating, Double</td>
</tr>
<tr>
<td></td>
<td></td>
<td>except BB, WW, LL, FF, DD</td>
</tr>
<tr>
<td>ADD, SUBtract, MULTIply, DIVide</td>
<td>Byte, Word, Longword, Floating, Double</td>
<td>2 operand, 3 operand</td>
</tr>
<tr>
<td>Bit Set, Bit Clear</td>
<td></td>
<td>Byte, Word, Longword</td>
</tr>
<tr>
<td>eXclusive OR</td>
<td></td>
<td>Floating, Double</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Extended MODulus, POLYnominal</td>
</tr>
</tbody>
</table>

Figure 6-1 Integer & Floating Instructions

<table>
<thead>
<tr>
<th>INSTRUCTION</th>
<th>DATA TYPE</th>
</tr>
</thead>
<tbody>
<tr>
<td>PUSH Longword</td>
<td></td>
</tr>
<tr>
<td>INCrement, DECrement</td>
<td>Byte, Word, Longword</td>
</tr>
<tr>
<td>MOVE, CLearR</td>
<td>Quadword</td>
</tr>
<tr>
<td>MOVE Zero-extend</td>
<td></td>
</tr>
<tr>
<td>ConVerT Round</td>
<td></td>
</tr>
</tbody>
</table>

122
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 values of the first two operands and stores the result in the third operand.

The integer optimizations include an instruction to push a longword onto the stack. Each integer data type includes operations for increment and decrement. VAX-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 modulus (range reduction) and polynomial calculation.
to aid in the implementation of mathematical functions. VAX-11 also includes shift and rotate instructions.

**FLOATING POINT INSTRUCTIONS**
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

\[\frac{1}{2} \text{ LEQ } f \text{ LSS } 1.\]

The fraction 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 arithmetic 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 overflow occurs.

Accuracy

General comments on the accuracy of the VAX-11 floating point instruction 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 prior accuracy of the operands is thus ignored. For all arithmetic operations, except DIV, a zero operand implies that the instruction is exact. The same statement 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 the ADD, SUB, MUL and DIV, and 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
   \[ \text{rounded value} = \text{chopped value} = \text{true value}. \]

2. If a stored result is not exact, its 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 and SUB may result in propagation of a carry, and hence the overflow bit is necessary. Second, if in ADD or SUB there is a one bit loss of significance in conjunction with an alignment shift of two or more bits, the first guard bit is needed for the LSB of the normalized result, and the second is then the rounding bit. So the three bits are necessary. A number of constraints must be observed in selection of the algorithms for the basic operations in order for these three bits to be sufficient to guarantee an error bound of \(\frac{1}{2}\) 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 \(\frac{1}{2}\) LSB. If, however, a negation follows the alignment shift, the error bound will be
     \[ \left(\frac{1}{2}\right) \times (1 + 2^{\text{g+2}}) \text{ LSB} \]
     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 (1/2) LSB.

2. MUL:
• The product of two normalized binary fractions can be as small as 1/4 and must be less than one. The overflow bit is not needed for MUL, but the first guard bit will be necessary for normalization if the product if less than 1/2, 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 (1/2) LSB for all floating 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.

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).
Integer and Floating Point Instructions

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 are used as 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.
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    MOV B  Move Byte
B0    MOV W  Move Word
D0    MOV L  Move Long
7D    MOV Q  Move Quad
50    MOV F  Move Floating
70    MOV D  Move Double

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

Notes:
1. On a floating reserved operand fault, the destination operand is unaffected and the condition codes are unpredictable.
2. Unlike the PDP-11, but like the other VAX-11 instructions, MOV B and MOV W do not modify the high order bytes of a register destination. Refer to the MOVZxL and CVT xL instructions to update the full register contents.
**PUSHL**

**Purpose:** push source operand onto stack

**Format:** opcode src.rl

**Operation:** `(SP) ← src;`

**Condition:**
- `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 shorter.

---

**PUSH LONG**
CLEAR

Purpose: clear a scalar quantity

Format: opcode dst.wx

Operation: dst ← 0;

Condition N ← 0;

Codes: Z ← 1;
        V ← 0;
        C ← C;

Exceptions: None

Opcodes:

<table>
<thead>
<tr>
<th></th>
<th>Opcode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>94</td>
<td>CLRB</td>
<td>Clear Byte</td>
</tr>
<tr>
<td>B4</td>
<td>CLRW</td>
<td>Clear Word</td>
</tr>
<tr>
<td>D4</td>
<td>CLRL</td>
<td>Clear Long</td>
</tr>
<tr>
<td>D4</td>
<td>CLRF</td>
<td>Clear Floating</td>
</tr>
<tr>
<td>7C</td>
<td>CLRQ</td>
<td>Clear Quad</td>
</tr>
<tr>
<td>7C</td>
<td>CLRD</td>
<td>Clear Double</td>
</tr>
</tbody>
</table>

Description: The destination operand is replaced by 0.

Notes: CLRx dst is equivalent to MOVx S↑#0,dst, but is shorter.
MNEG

MOVE NEGATED

Purpose: move the arithmetic negation of a scalar quantity

Format: opcode src.rx, dst.wx

Operation: dst ← -scr;

Condition: N ← dst LSS0;

Codes: 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 the logical complement of an integer

Format: opcode src.rx, dst.wx

Operation: dst ← NOT src;

Condition: N ← dst LSS 0;

Codes: 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.
**Integer and Floating Point Instructions**

**CVT**

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

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

**Operation:**  dst ← conversion of src;

**Condition:**  N ← dst LSS 0;

**Codes:**  Z ← dst EQL 0;

V ← |src cannot be represented in dst|;

C ← 0;

**Exceptions:**

- Integer overflow
- Floating overflow
- Reserved operand

**Opcodes:**

<table>
<thead>
<tr>
<th>OPCODE</th>
<th>Opcode</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>99</td>
<td>CVTBD</td>
<td>Convert Byte to Word</td>
</tr>
<tr>
<td>98</td>
<td>CVTBL</td>
<td>Convert Byte to Long</td>
</tr>
<tr>
<td>33</td>
<td>CVTWF</td>
<td>Convert Word to Byte</td>
</tr>
<tr>
<td>32</td>
<td>CVTLD</td>
<td>Convert Word to Long</td>
</tr>
<tr>
<td>F6</td>
<td>CVTTL</td>
<td>Convert Long to Byte</td>
</tr>
<tr>
<td>F7</td>
<td>CVTTP</td>
<td>Convert Long to Word</td>
</tr>
<tr>
<td>4C</td>
<td>CVTF</td>
<td>Convert Byte to Floating</td>
</tr>
<tr>
<td>C</td>
<td>CVTFW</td>
<td>Convert Byte to Double</td>
</tr>
<tr>
<td>C</td>
<td>CVTFD</td>
<td>Convert Word to Double</td>
</tr>
<tr>
<td>4D</td>
<td>CVTLD</td>
<td>Convert Long to Floating</td>
</tr>
<tr>
<td>4E</td>
<td>CVTMD</td>
<td>Convert Long to Double</td>
</tr>
<tr>
<td>68</td>
<td>CVTMD</td>
<td>Convert Double to Byte</td>
</tr>
<tr>
<td>49</td>
<td>CVTFW</td>
<td>Convert Double to Word</td>
</tr>
<tr>
<td>69</td>
<td>CVTMD</td>
<td>Convert Double to Long</td>
</tr>
<tr>
<td>4A</td>
<td>CVFNL</td>
<td>Convert Floating to Long</td>
</tr>
<tr>
<td>4B</td>
<td>CVFNL</td>
<td>Convert Rounded Floating to Long</td>
</tr>
<tr>
<td>6A</td>
<td>CVFNL</td>
<td>Convert Double to Long</td>
</tr>
<tr>
<td>6B</td>
<td>CVFNL</td>
<td>Convert Rounded Double to Long</td>
</tr>
<tr>
<td>56</td>
<td>CVFNL</td>
<td>Convert Floating to Double</td>
</tr>
<tr>
<td>76</td>
<td>CVFNL</td>
<td>Convert Double to Floating</td>
</tr>
</tbody>
</table>

**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
- CVTBD: exact
- CVTWF: exact
- CVTWD: exact
- CVTFW: truncated
- CVTDF: truncated
- CVTFD: truncated
- CVTRFL: rounded
Integer and Floating Point Instructions

CVTTF rounded CVTDL truncated
CVTLDF 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 results.
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 in unaffected and the condition codes are unpredictable.

Example:
CONVERT FLOATING TO WORD
CVTFW WORK, R0 ;Convert contents of WORK
;floating to word
;store in R0

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

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

Example:
CONVERT ROUNDED FLOATING TO LONG
CVTRFL R2,R3 ;Converts contents of R2
;floating to long, rounding
;store in R3

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

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

**MOVE ZERO-EXTENDED**

**Purpose:** convert an unsigned integer to a wider unsigned integer

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

**Operation:** \[ \text{dst} \leftarrow \text{ZEXT} (\text{src}); \]

**Condition:** \[ N \leftarrow 0; \]

**Codes:** \[ Z \leftarrow \text{dst} \text{ EQL} 0; \]
\[ V \leftarrow 0; \]
\[ C \leftarrow C; \]

**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: N ← src1 LSS src2;

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

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

Opcodes:

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>91</td>
<td>CMPB</td>
<td>Compare Byte</td>
</tr>
<tr>
<td>B1</td>
<td>CMPW</td>
<td>Compare Word</td>
</tr>
<tr>
<td>D1</td>
<td>CMPL</td>
<td>Compare Long</td>
</tr>
<tr>
<td>51</td>
<td>CMPF</td>
<td>Compare Floating</td>
</tr>
<tr>
<td>71</td>
<td>CMPD</td>
<td>Compare Double</td>
</tr>
</tbody>
</table>

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.
**INC**

**Purpose:** add 1 to an integer

**Format:** opcode sum.mx

**Operation:** sum ← sum + 1;

**Condition:** N ← sum LSS 0;

**Codes:** 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 is incremented. On overflow, the sum operand is replaced by the largest negative integer.

2. INCx sum is equivalent to ADDx2 #1, sum, but is shorter.
TEST

Purpose: arithmetic compare of a scalar to 0.

Format: opcode src.rx

Operation: src−0;

Condition: N ← src LSS 0;

Codes: Z ← src EQL 0;

V ← 0;

C ← 0;

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

Opcodes:

<table>
<thead>
<tr>
<th>Code</th>
<th>Opcode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>95</td>
<td>TSTB</td>
<td>Test Byte</td>
</tr>
<tr>
<td>B5</td>
<td>TSTW</td>
<td>Test Word</td>
</tr>
<tr>
<td>D5</td>
<td>TSTL</td>
<td>Test Long</td>
</tr>
<tr>
<td>53</td>
<td>TSTF</td>
<td>Test Floating</td>
</tr>
<tr>
<td>73</td>
<td>TSTD</td>
<td>Test Double</td>
</tr>
</tbody>
</table>

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
opcode add1.rx, add2.rx, sum.wx

**Operation:**
sum ← sum + add;
sum ← add1 + add2;

**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.
Integer and Floating Point Instructions

Example: ADD FLOATING 2 OPERAND
ADDF2 #144., WORK

;ADD 144 floating point
;format to WORK

Initial Conditions:
WORK = 00000000

After Instruction Execution:
WORK = 00004410

Example: ADD FLOATING 3 OPERAND
ADDF3 #144., WORK, WORK1

;Add 144 Floating
;point format to contents
;of WORK; store result
;in WORK1

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

After Instruction Execution:
WORK = 00004410

WORK1 = 00004490 (hex); (288 floating)
ADWC

**Purpose:** perform extended-precision addition

**Format:** opcode add.rl, sum.ml

**Operation:** sum ← sum + add + C;

**Condition**
N ← sum LSS 0;
Z ← sum EQL 0;
V ← [integer overflow];
C ← [carry from most significant bit];

**Codes:**

**Exceptions:** Integer overflow

**Opcodes:** D8 ADWC Add with Carry

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

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

**Example:** ADD WITH CARRY

To add two quadword integers:
ADDL A, B ;add low half
ADWC A+4, B+4 ;including carry

Additional ADWC can be appended for greater precision.
ADD ALIGNED WORD INTERLOCKED

Purpose: maintain operating system resource usage counts

Format: opcode add.rw, sum.mw

Operation: 
\[
tmp \leftarrow \text{add}; \\
\{\text{set interlock}\}; \\
\text{sum} \leftarrow \text{sum} + \text{tmp}; \\
\{\text{released interlock}\}; \\
\]

Condition: \(N \leftarrow \text{sum LSS 0};\)

Codes: \(Z \leftarrow \text{sum EQL 0};\)
\(V \leftarrow \text{|integer overflow|};\)
\(C \leftarrow \text{|carry from most significant bit|};\)

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 inter-
locked against ADAWI operations by other processors or de-
vices in the system. The destination must be aligned on a word
boundary i.e., bit zero of the sum operand address must be
zero. If it is not, a reserved operand fault is taken.

Notes:
1. Integer overflow occurs if the input operands to the add
have the same sign and the result has the opposite sign.
On overflow, the sum operand is replaced by the low or-
der bits of the true result.

2. If the addend and the sum operand overlap, the result
and the condition codes are UNPREDICTABLE.
SUB

**Purpose:** perform arithmetic subtraction

**Format:**
- opcode sub.rx, dif.mx  
- opcode sub.rx, min.rx, dif.wx  
  2 operand
  3 operand

**Operation:**
- dif ← dif - sub;  
- dif ← min - sub;  
  2 operand
  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 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
DEC

**Purpose:** subtract 1 from an integer

**Format:** opcode dif.mx

**Operation:** \( \text{dif} \leftarrow \text{dif} - 1; \)

**Condition**
- \( N \leftarrow \text{dif LSS 0}; \)
- \( Z \leftarrow \text{dif EQL 0}; \)
- \( V \leftarrow \{\text{integer overflow}\}; \)
- \( C \leftarrow \{\text{borrow from most significant bit}\}; \)

**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 SUBx2 #1, dif, but is shorter.
SUBTRACT WITH CARRY

Purpose: perform extended-precision subtraction
Format: opcode sub.rl, dif.ml
Operation: \( \text{dif} \leftarrow \text{dif} - \text{sub} - C \)
Condition: \( \text{N} \leftarrow \text{dif} \text{LSS} 0 \)
Codes: \( \text{Z} \leftarrow \text{dif} \text{EQL} 0 \); \( \text{V} \leftarrow \text{integer overflow} \);
\( \text{C} \leftarrow \text{borrow from most significant bit} \)
Exceptions: Integer overflow
Opcodes: D9 SBWC Subtract with Carry
Description: The subtrahend operand and the contents of the condition code C bit are subtracted from the difference operand and the difference operand is replaced by the result.
Notes: 1. On overflow, the difference operand is replaced by the low order bits of the true result.
2. The two subtractions in the operation are performed simultaneously.

Example: SUBTRACT WITH CARRY
To subtract two quadword integers:
SUBL A, B ;subtract low half
SBWC A+4, B+4 ;subtract high half including ;borrow
Additional SBWC can be appended for greater precision.
MUL

Purpose: perform arithmetic multiplication

Format: opcode mulr.rx, prod.mx
opcode mulr.rx, muld.rx, prod.wx

Operation: prod ← prod * mulr;
prod ← muld * mulr;

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

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

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

**Purpose:** perform extended-precision multiplication

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

**Operation:** prod ← [muld * mulr] + SEXT(add);

**Condition:** N ← prod LSS 0;
Z ← prod EQS 0;
V ← 0;
C ← 0;

**Exceptions:** None

**Opcodes:** 7A EMUL

**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.

**Notes:**

**Example:**

EXTENDED MULTIPLY

To multiply two quadwords, producing a quadword;

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

Condition N ← fract LSS 0;
Codes:    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 multiplier (first operand) to gain eight 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 (before normalization) 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.
DIV

**Purpose:** perform arithmetic division

**Format:**
- opcode divr.rx, quo.mx
- opcode divr.rx, divd.rx, quo.wx

**Operation:**
- quo ← quo / divr;
- quo ← divd / divr;

**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. In 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).

\[ N \leftarrow 1; Z \leftarrow 0; V \leftarrow 1; \text{ and } C \leftarrow 0. \]

**Example:**

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

;Divide

**Initial Conditions:**

- \( R4 = 00004100 \)
- \( R2 = 00004330 \)

**After Instruction Execution:**

- \( R4 = 00004100 \)
- \( R2 = 000042B0 \)
EDIV

EXTENDED DIVIDE

Purpose: perform extended-precision division
Format: opcode divr.rl, divd.rq, quo.wl, rem.wl
Operation: quo ← divd/divr;
rem ← REM{divd, divr};
Condition
N ← quo LSS 0;
Z ← quo EQL 0;
V ← {integer overflow} OR {divr EQL 0};
C ← 0;
Codes:
Exceptions: Integer overflow
Divide by zero
Opcodes: 7B EDIV
Description: The dividend operand is divided by the divisor operand; the
quotient operand is replaced by the quotient and the remain-
der 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, or if the divisor operand is 0, then the quo-
tient operand is replaced by bits 31:0 of the dividend op-
erand, and the remainder is replaced by 0.
BIT TEST

Purpose: test a set of bits for all zero

Format: opcode mask.rx, src.rx

Operation: tmp ← src AND mask;

Condition: N ← tmp LSS 0;

Codes: Z ← tmp EQL 0;

V ← 0;

C ← C;

Exceptions: None

Opcodes: 93 BITB Bit Test Byte

B3 BITW Bit Test Word

D3 BITL Bit Test Long

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

**Purpose:** perform logical inclusive OR of two integers

**Format:**
- opcode mask.rx, dst.mx
- opcode mask.rx, src.rx, dst.wx

**Operation:**
- dst ← dst OR mask;
- dst ← src OR mask;

**Condition:**
- N ← dst LSS 0;

**Codes:**
- 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  
        opcode mask.rx, src.rx, dst.wx  
        2 operand  
        3 operand

Operation: dst ← dst AND [NOT mask];  
           dst ← src AND [NOT mask];  
           12 operand  
           13 operand

Condition: N ← dst LSS 0;

Codes: 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 1's ones complement of the mask operand and the destination operand is replaced by the result.
**XOR**

**Purpose:** perform logical exclusive OR of two integers

**Format:**
- opcode mask.rx, dst.mx
- opcode mask.rx, src.rx, dst.wx

**Operation:**
- dst ← dst XOR mask;
- dst ← src XOR mask;

**Condition:**
- N ← dst LSS 0;

**Codes:**
- 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: $dst \leftarrow src$ shifted cnt bits;
Condition: $N \leftarrow dst$ LSS 0;
Codes: $Z \leftarrow dst$ EQL 0;
$V \leftarrow \{integer\ \overline{\text{overflow}}\};$
$C \leftarrow 0;$
Exceptions: integer overflow
Opcodes: 78 ASHL Arithmetic Shift Long
97 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 zero count operand replaces the destination operand with the unshifted source operand.
Notes:
1. Integer overflow occurs on a left shift if any bit shifted into the sign bit position differs from the sign bit of the source operand. On overflow, the destination operand is replaced by the low order bits of the true result.
2. If cnt GTR 32 (ASHL) or cnt GTR 64 (ASHQ); the destination operand is replaced by 0
3. If cnt LEQ −32 (ASHL) or cnt LEQ −63 (ASHQ); all the bits of the destination operand are copies of the sign bit of the source operand.
4. A left shift is equivalent to a multiply by the corresponding power of two. A right shift is not, however, equivalent to a divide because negative numbers are rounded away from zero.
ROTATE LONG

Purpose: rotate of integer
Format: opcode cnt.rb, src.rl, dst.wl
Operation: dst ← src rotated cnt bits;
Condition: N ← dst LSS 0;
Codes: 
   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.rw, tbladdr.ab

Operation: \[
\begin{align*}
C_{\text{degree}} & \rightarrow \text{tbladdr} \\
C_1 & \\
C_0 & \\
\text{result} & \leftarrow C_{\text{degree}}
\end{align*}
\]

For degree times, loop

result ← arg * result;

!Perform multiply, and retain an
!extended floating fraction of
!31 bits (POLYF) or 63 bits (POLYD)
!the fraction is truncated before
!normalization)
!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:

\[
\text{result} = C[0] + x^*(C[1] + x^*(C[2] + \ldots x^*(C[d])))
\]

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

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:
   - If PSL<FPD> = 0, the reserved operand is either the degree operand (greater than 31), or the argument operand, or some coefficient.
   - If PSL<FPD> = 1, the reserved operand is a coefficient, and R3 is pointing at the value which caused the exception.
   - The state of the saved condition codes and the other registers is unpredictable. If the reserved operand is changed and the contents of the condition codes and all registers are preserved, the fault is continu-able.

6. On floating underflow after the rounding operation, the temporary result 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.

To compute \( P(x) = C_0 + C_1 x + C_2 x^2 \)
where \( C_0 = 1.0, C_1 = .5, \) and \( C_2 = .25 \)

```
POLYF X,#2,PTABLE
```

```
PTABLE:

.FLOT 0.25 ;C2
.FLOT 0.5 ;C1
.FLOT 1.0 ;C0
```
CHAPTER 7
SPECIAL INSTRUCTIONS

INTRODUCTION
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.

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 PUSH instruction saves multiple registers by pushing them onto the stack. The POP 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 standard for calling and signaling requires that registers be saved in the call frame (see Appendix C and Chapter 8). Thus, any registers manipulated by PUSH or POP, except R0 and R1, must appear in the procedure entry mask.

The VAX software standard for calling and signalling require any registers between R2 and R11 which are modified by the procedure to be saved in the call frame by setting up the appropriate entry mask. R0 and R1 are used to return procedure status. PUSH/POP should be used to save and restore only those registers specified in the procedure entry mask. If a procedure saves registers not noted in the entry mask and it gets an exception, its caller's registers cannot be restored properly by the unwinding mechanism.
Special Instructions

PUSHR

PUSH REGISTERS

Purpose: save multiple registers or stack

Format: opcode mask.rw

Operation:

\[ \text{SP AFTER} \]

\[ \text{SAVED REGISTERS IN ORDER} \]

\[ R_0 \ldots \ldots R_{14} \]

\[ \text{SP BEFORE} \]

Condition: \( N \leftarrow N; \)

Codes: \( Z \leftarrow Z; \)

\( V \leftarrow V; \)

\( C \leftarrow C; \)

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:

1. 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.

2. This instruction is similar to the sequence

\[
\begin{align*}
PUSHL & \quad R_{14} \\
PUSHL & \quad R_{13} \\
& \quad \vdots \\
PUSHL & \quad R_0
\end{align*}
\]

where only the masked registers are pushed.
**Special Instructions**

**Example:**

```
PUSHR #\[M< R0, R1, R2, R3>
```

; saves R0 through R3

**Example:**

```
PUSHR #\[M< R1, R6, R7>
```

; saves R1, R6, and R7
**Special Instructions**

**POPR**

**Purpose:** restore multiple registers from stack

**Format:** opcode mask.rw

**Operation:**

```
  SAVED
  REGISTERS
  IN ORDER
  R0......R14

  ← SP BEFORE

  ← SP AFTER
```

**Condition:** N ← N;

**Codes:**
- Z ← Z;
- V ← V;
- C ← C;

**Exceptions:** None

**Opcodes:**
- BA       POPR

**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.

**Notes:** This instruction is similar to the sequence

```
MOVL   (SP)+,R0
MOV L   (SP)+,R1
...
...
MOV L   (SP)+,R14
```

where only the masked registers are popped.
PROCESSOR STATUS LONGWORD MANIPULATION

MOVE FROM PSL

Purpose: obtain processor status
Format: opcode dst.wl
Operation: dst ← PSL;
Condition: N ← N;
Codes: Z ← Z;
V ← V;
C ← C;
Exceptions: none
Opcodes: DC MOVPSL Move from PSL

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

BISPSW
BICPSW

Purpose: set or clear trap enables

Format: opcode mask.rw

Operation: PSW ← PSW OR mask;
PSW ← PSW AND {NOT mask};

!BISPSW
!BICPSW

Condition Codes:

N ← N OR mask <3>;
Z ← Z OR mask <2>;
V ← V OR mask <1>;
C ← C OR mask <0>;
N ← N AND {NOT mask} <3>;
Z ← Z AND {NOT mask} <2>;
V ← V AND {NOT mask} <1>;
C ← C AND {NOT mask} <0>;

!BISPSW
!BICPSW

Exceptions: Reserved Operand

Opcodes: B8 BISPSW Bit set PSW
B9 BICPSW Bit clear PSW

Description: On BISPSW, the processor status longword is ORed with the 16-bit mask operand and the PSW is replaced by the result. On BICPSW, the processor status longword is ANDed with the 1's complement of the 16-bit mask operand and the PSW is replaced by the result.

Notes: A reserved operand fault occurs if mask <15:8> is not zero. On a reserved operand fault, the PSW is not affected.

Example: BISPSW #M<FU> ; enables floating underflow traps
ADDRESS INSTRUCTIONS

MOVE ADDRESS
PUSH ADDRESS

Purpose: calculate address of quantity

Format: opcode src.ax, dst.wl
opcode src.ax

Operation: dst ← src;
−(SP) ← src;

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.
Special Instructions

3. The only difference between the MOVAx\$ (PUSHx\$) is the context of the src. This only affects autoincrement, autodecrement, and indexing.

Example: PUSHAL XYZ ; pushes the address of longword XYZ
INDEX INSTRUCTION
The index instruction (INDEX) calculates an index for an array of fixed length data types (integer and floating) and for arrays of bit fields, character strings, and decimal strings. It accepts as arguments: a subscript, lower and upper subscript bounds, an array element size, a given index, and a destination for the calculated index. It incorporates range checking within the calculation for high-level languages using subscript bounds, and it allows index calculation optimization by removing invariant expressions.
Special Instructions

INDEX

Purpose: index calculation of arrays of fixed length data, bit fields, and strings

Format: opcode subscript.rl, low.rl, high.rl, size.rl, indexin.rl, indexout.wl

Operation: indexout ← [indexin + subscript] *size;
if [subscript LSS low] or [subscript GTR high]
then [subscript range trap];

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

Exceptions: subscript range

Opcodes: 0A INDEX Index

Description: The indexin operand is added to the subscript operand and the sum is multiplied by the size operand. The indexout operand is replaced by the result. If the subscript operand is less than the low operand or greater than the high operand, a subscript range trap is taken.

Notes: 1. No arithmetic exception other than subscript range can result from this instruction. Thus no indication is given if overflow occurs in either the add or multiply steps. If overflow occurs on the add step the sum is the low order 32 bits of the true result. If overflow occurs on the multiply step the indexout operand is replaced by the low order 32 bits of the true product of the sum and the subscript operand. In the normal use of this instruction, overflow cannot occur without a subscript range trap occurring.

2. The index instruction is useful in index calculations for arrays of the fixed length data types (integer and floating) and for index calculations for arrays of bit fields, character strings, and decimal strings. The indexin operand permits cascading INDEX instructions for multidimensional arrays. For one-dimensional bit field arrays it also permits introduction of the constant portion of an index calculation which is not readily absorbed by address arithmetic.

174
3. This instruction is similar to the sequence
   ADDL3 indexin, subscript, indexout
   MULL2 size, indexout
   CMPL subscript, low
   BLSS 10$
   CMPL subscript, high
   BLEQ 20$
   10$: [trap error]
   20$:

Example:

The COBOL statements:
01 A-ARRAY.
   02 A PIC x(10) occurs 15 times
01 B PIC x(10).
   MOVE A(I) to B.
are equivalent to:
   INDEX I, #1, #15, #10, #0, R0
   MOVC3 #10, A-10[R0], B.

The PL/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
**QUEUE INSTRUCTIONS**

A queue is a circular, doubly linked list. A queue entry is specified by its address. Each queue entry is linked to the next via a pair of longwords. The first longword is the forward link (FLINK): it specifies the location of the succeeding entry. The second longword is the backward link (BLINK): it specifies the location of the preceding entry. VAX-11 supports two distinct types of links: absolute, and self-relative. An absolute link contains the absolute address of the entry that it points to. A self-relative link contains a displacement from the present queue entry. A queue is classified by the type of link it uses.

**Absolute Queues**

Absolute queues use absolute addresses as links. Queue entries are linked by 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.

The following examples illustrates queue operations, an empty queue is specified by its header at address H:

```
<table>
<thead>
<tr>
<th>31</th>
<th></th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>H</td>
<td>: H</td>
</tr>
<tr>
<td></td>
<td>H</td>
<td>: H+4</td>
</tr>
</tbody>
</table>
```

If an entry at address B is inserted into an empty queue (at either the head or tail) the queue is as shown below:

```
<table>
<thead>
<tr>
<th>31</th>
<th></th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>B</td>
<td>: H</td>
</tr>
<tr>
<td></td>
<td>B</td>
<td>: H+4</td>
</tr>
</tbody>
</table>
```
If an entry at address A is inserted at the head of the queue, the queue is as shown below:

Finally, if an entry at address C is inserted at the tail, the queue appears as follows:
Following the above steps in reverse order gives the effect of removal at the tail and removal at the head.
If more than one process can perform operations on a queue simultaneously, insertions and removals should only be done at the head or tail of the queue. If only one process (or one 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 above example with the queue containing entries A, B, and C, the entry at address B can be removed giving:

The reason for the above restriction is that operations at the head or tail are always valid because the queue header is always present; operations elsewhere in the queue depend on specific entries being present and may become invalid if another process is concurrently performing operations on the queue.

Two instructions are provided for manipulating absolute queues: INSQUE, and REMQUE. INSQUE inserts an entry specified by an entry operand into the queue following the entry specified by the predecessor operand. REMQUE removes the entry specified by the entry operand. Queue entries can be on arbitrary byte boundaries. Both INSQUE and REMQUE are implemented as non-interruptible instructions.
INSQUE

INSERT ENTRY IN QUEUE

Purpose: add entry to head or tail of queue
Format: opcode entry.ab, pred.ab
Operation: If all memory accesses can be completed then

begin
(entry) ← (pred); !forward link of entry
(entry+4) ← pred; !backward link of entry
((pred)+4) ← entry; !backward link of successor
(pred) ← entry; !forward link of predecessor
end;

else

begin
[backup instruction];
[initiate fault];
end;

Condition: N ← (entry) LSS (entry+4);
Codes: Z ← (entry) EQL (entry+4); !first entry in queue
V ← 0;
C ← (entry) LSSU (entry+4);

Exceptions: None

Opcodes: 0E INSQUE Insert Entry in Queue

Description: The entry specified by the entry operand is inserted into the queue following the entry specified by the predecessor operand. If the entry inserted was the first one in the queue, the condition code Z-bit is set; otherwise it is cleared. The insertion is a 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 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 in a single processor may access a shared list without additional synchronization if the insertions and removals are only at the head or trail of the queue.

3. During access validation, any access which cannot be completed results in a memory management exception even though the queue insertion is not started.
4. This instruction is similar to the interlocked sequence

\[
\begin{align*}
\text{MOVL} & \quad \text{pred, temp reg} \\
\text{MOVAB} & \quad \text{pred, entry+4} \\
\text{MOVAB} & \quad \text{entry, 4 (temp reg)} \\
\text{MOVL} & \quad \text{temp reg, entry} \\
\text{MOVAB} & \quad \text{entry, pred}
\end{align*}
\]

**Example:** Three types of insertion can be performed by appropriate choice of predecessor operand:

Insert at head

\[
\text{INSQUE entry,h ;h is queue head}
\]

Insert at tail

\[
\text{INSQUE entry,@h+4 ;h is queue head}
\]

(Note "@" in this case only)

Insert after arbitrary predecessor

\[
\text{INSQUE entry,p ;p is predecessor}
\]

To set a software interlock realized with a queue, the following can be used:

\[
\begin{align*}
\text{INSQUE ... ;was queue empty?} \\
\text{BEQL 1$ ;yes} \\
\text{CALL WAIT(...) ;no, wait} \\
1$:
\end{align*}
\]
**REMQUE**

**REMOVE ENTRY FROM QUEUE**

**Purpose:** remove entry from head or tail of queue

**Format:** opcode entry.ab, addr.wl

**Operation:**
if all memory accesses can be completed then
begin
((entry+4) ← (entry); !forward link of predecessor
((entry)+4) ← (entry+4); !backward link of successor
addr ← entry;
end;
else
begin
[backup instruction];
[initiate fault];
end;

**Condition:** N ← (entry) LSS (entry+4);

**Codes:**
Z ← (entry) EQL (entry+4); !removed last entry
V ← entry EQL (entry+4); !no entry to remove
C ← (entry) LSSU (entry+4);

**Exceptions:** None

**Opcodes:** 0F REMQUE Remove Entry from Queue

**Description:** The queue entry specified by the entry operand is removed from the queue. The address operand is replaced by the address of the entry removed. If there was no entry in the queue to be removed, the condition code V bit is set; otherwise it is cleared. If the queue is empty at the end of this instruction, the condition code Z-bit is set; otherwise it is cleared. The removal is a 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 in a single processor may access a shared list without additional synchronization if insertions and removals are only at the head or tail of the queue.
3. During access validation, any access which cannot be completed results in a memory management exception even though the queue removal is not started.
4. This instruction is similar to the interlocked sequence

\[
\begin{array}{ll}
\text{MOVL} & \text{entry, } @\text{entry}+4 \\
\text{MOVL} & \text{entry, tempreg} \\
\text{MOVL} & \text{entry+4, 4 (tempreg)} \\
\text{MOVAB} & \text{entry, addr}
\end{array}
\]

**Example:** Three types of removal can be performed by suitable choice of entry operand:

Remove at head
\[
\text{REMQUE} \quad @h, \text{addr} \quad ; h \text{ is queue header}
\]

Remove at tail
\[
\text{REMQUE} \quad @h+4, \text{add} \quad ; h \text{ is queue header}
\]

Remove arbitrary entry
\[
\text{REMQUE} \quad \text{entry,addr} \quad ;
\]

To release a software interlock realized with a queue, the following can be used:

\[
\begin{array}{ll}
\text{REMQUE ...} & ; \text{queue empty?} \\
\text{BEQL} \quad 1$ & ; \text{yes} \\
\text{CALL} \quad \text{ACTIVATE (...)} & ; \text{activate other waiters}
\end{array}
\]

1$:

To remove entries until the queue is empty, the following can be used:

\[
\begin{array}{ll}
\text{1$: REMQUE...} & ; \text{anything removed?} \\
\text{BVS} \quad \text{EMPTY} & ; \text{no}
\end{array}
\]

... 

\[
\text{BR} \quad 1$
\]

;
Self-Relative Queues
Self-relative queues use displacements from queue entries as links. Queue entries are linked by a pair of longwords. The first longword (lowest addressed) is the forward link; displacement of the succeeding queue entry from the present entry. The second longword (highest addressed) is the backward link; the displacement of the preceding queue from the present entry. A queue is specified by a queue header, which also consists of two longword links.

The following contains examples of queue operations. An empty queue is specified by its header at address H. Since the queue is empty, the self-relative links must be zero as shown below:

```
<table>
<thead>
<tr>
<th>31</th>
<th>0</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>H</td>
<td>H+4</td>
<td></td>
</tr>
</tbody>
</table>
```

If an entry at address B is inserted into an empty queue (at either the head or tail), the queue is as shown below:

```
<table>
<thead>
<tr>
<th>31</th>
<th>B-H</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>B-H</td>
<td></td>
</tr>
<tr>
<td>H</td>
<td>H+4</td>
<td></td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>31</th>
<th>H-B</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>H-B</td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>B+4</td>
<td></td>
</tr>
</tbody>
</table>
```
Special Instructions

If an entry at address A is inserted at the head of the queue, the queue is as shown below:

\[ \begin{align*}
  &31 & &0 \\
  & & &H \\
  & & &H + 4 \\
  &31 & &0 \\
  & & &A \\
  & & &A + 4 \\
  &31 & &0 \\
  & & &B \\
  & & &B + 4 \\
  &31 & &0 \\
  & & &A - B \\
  & & &H - A \\
  & & &B - A \\
  & & &A - H \\
\end{align*} \]
Finally, if an entry at address C is inserted at the tail, the queue appears as follows:

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

Four operations can be performed on self-relative queues; insert at head, insert at tail, remove from head, and remove from tail. Furthermore, these operations are interlocked to allow cooperating processes in a multiprocessor system to access a shared list without additional synchronization. Queue entries must be quadword aligned.
Hardware supported interlocked memory access mechanism is used to read the queue header. Bit 0 of the queue header is used as a secondary interlock and is set when the queue is being accessed. If an interlocked queue instruction encounters the secondary interlock set, it terminates after setting the condition codes to indicate failure to gain access to the queue. If the secondary interlock bit is not set then the interlocked queue instruction sets it during its operation and clears it at instruction completion. This prevents other interlocked queue instructions from operating on the same queue.
INSQHI

INSERT ENTRY INTO QUEUE AT HEAD, INTERLOCKED

Format:    opcode entry.ab, header.aq

Condition Codes:
if [insertion succeeded] then
    begin
        N ← 0;
        Z ← (entry) EQL (entry + 4);  !first entry in queue
        V ← 0;
        C ← 0;
        end;
else
    begin
        N ← 0;
        Z ← 0;
        V ← 0;
        C ← 1;  !secondary interlock failed
        end;

Exceptions: reserved operand

Opcodes:    5C  INSQHI    Insert Entry into Queue at Head,
            Interlocked

Description: The entry specified by the entry operand is inserted into the
queue following the header. If the entry inserted was the first
one in the queue, the condition code Z-bit is set; otherwise it is
cleared. The insertion is a non-interruptible operation. The in-
sertion is interlocked to prevent concurrent interlocked inser-
tions or removals at the head or tail of the same queue by
another process even in a multiprocessor environment. Before
performing any part of the operation, the processor validates
that the entire operation can be completed. This ensures that if
a memory management exception occurs, the queue is left in a
consistent state. If the instruction fails to acquire the secondary
interlock, the instruction sets condition codes and terminates.

Notes:
1. Because the insertion is non-interruptible, processes run-
ning in kernel mode can share queues with interrupt ser-
vice routines.
2. The INSQHI, INSQTI, REMQHI, and REMQTI instructions
are implemented such that cooperating software processes in a multiprocessor may access a shared list
without additional synchronization.
Special Instructions

3. To set a software interlock realized with a queue, the following can be used:

```
INSERT: INSQHI ... ; was queue empty?
        BEQL 1$ ; yes
        BCS INSERT ; try inserting again
        CALL WAIT(...) ; no, wait
1$:
```

4. During access validation, any access which cannot be completed results in a memory management exception even though the queue insertion is not started.

5. A reserved operand fault occurs if entry or header is an address that is not quadword aligned (i.e., <2:0>NEQU 0) or if (header)<2:1> is not zero. A reserved operand fault also occurs if header equals entry. In this case the queue is not altered.
INSQTI

INSERT ENTRY INTO QUEUE AT TAIL, INTERLOCKED

Format: opcode entry.ab, header.aq

Condition Codes:
if [insertion succeeded] then
   begin
      N ← 0;
      Z ← (entry) EQL (entry+4) !first entry in queue
      V ← 0;
      C ← 0;
   end;
else
   begin
      N ← 0;
      Z ← 0;
      V ← 0;
      C ← 1; !secondary interlock failed
   end;

Exceptions: reserved operand

Opcodes: 5D INSQTI Insert Entry into Queue at Tail, Interlocked

Description: The entry specified by the entry operand is inserted into the queue preceding the header. 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 operation. The insertion is interlocked to prevent concurrent interlocked insertions or removals at the head or tail of the same queue by another process when in a multiprocessor environment. Before performing any part of the operation, the processor validates that the entire operation can be completed. This ensures that if a memory management exception occurs, the queue is left in a consistent state. If the instruction fails to acquire the secondary interlock, the instruction sets condition codes and terminates.

Notes:
1. Because the insertion is non-interruptible, processes running in kernel mode can share queues with interrupt service routines.
2. The INSQHI, INSQTI, REMQHI, and REMQTI instructions are implemented such that cooperating software processes in a multiprocessor may access a shared list without additional synchronization.
3. To set a software interlock realized with a queue, the following can be used:

```
INSERT:       INSQHI ... ; was queue empty?
    BEQL    $1$ ; yes
    BCS    INSERT ; try inserting again
    CALL    WAIT(...) ; no, wait
$1$;
```

4. During access validation, any access which cannot be completed results in a memory management exception even though the queue insertion was not started.

A reserved operand fault occurs if entry, header, or (header+4) is an address that is not quadword aligned (i.e., $<$2:0$>$ NEQU 0) or if (header)$<$2:1$>$ is not zero. A reserved operand fault also occurs if header equals entry. In this case the queue is not altered.
**REMQHI**

**REMOVE ENTRY FROM QUEUE AT HEAD, INTERLOCKED**

**Format:** opcode header.aq, addr.wl

**Condition**

if [removal succeeded] then

```
begin
N ← 0;
Z ← (header) EQL 0; !queue empty
V ← tmp1 EQL 0; !no entry to remove
C ← 0;
end;
```

else

```
begin
N ← 0;
Z ← 0;
V ← 1; !did not remove anything
C ← 1; !secondary interlock failed
end;
```

**Exceptions:** reserved operand

**Opcodes:** 5E REMQHI Remove Entry from Queue at Tail, Interlocked

**Description:**
The queue entry following the header is removed from the queue. The address operand is replaced by the address of the entry removed. If no entry was removed from the queue (because either there is nothing to remove or secondary interlock failed), the condition code V bit is set; otherwise it is cleared. If the interlock succeeded and the queue is empty at the end of this instruction, the condition code Z-bit is set; otherwise it is cleared. The removal is interlocked to prevent concurrent interlocked insertions or removals at the head or tail of the same queue by another process even in a multiprocessor environment. The removal is a 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. If the instruction fails to acquire the secondary interlock, the instruction sets condition codes and terminates without altering the queue.

**Notes:**

1. Because the removal is non-interruptible, processes running in kernel mode can share queues with interrupt service routines.
2. The INSEQH, INSEQTI, REMQHI, and REMQT1 instructions are implemented such that cooperating software processes in a multiprocessor may access a shared list without additional synchronization.

3. To release a software interlock realized with a queue, the following can be used:

$$1$: REMQHI ...
   \text{;removed last?}
   \text{BEQL 2}$
   \text{;yes}
   \text{BCS 1}$
   \text{;try removing again}
   \text{CALL ACTIVATE(...)}
   \text{;Activate other waiters}

$$2$: 

4. To remove entries until the queue is empty, the following can be used:

$$1$: REMQHI ...
   \text{;anything removed?}
   \text{BVS 2}$
   \text{;no}
   \text{.}
   \text{process removed entry}
   \text{.}
   \text{BR 1}$

$$2$: BCS 1$
   \text{;try removing again}
   \text{queue empty}

5. During access validation, any access which cannot be completed results in a memory management exception even through the queue removal is not started.

6. A reserved operand fault occurs if header or (header + (header)) is an address that is not quadword aligned (i.e. $<2:0>$ NEQU 0) or if (header)$<2:1>$ is not zero. A reserved operand fault also occurs if header equals addr. In this case the queue is not altered.
Special Instructions

REMQTI

REMOVE ENTRY FROM QUEUE AT TAIL, INTERLOCKED

Format:  opcode header.aq, addr.wl

Condition Codes: if [removal succeeded] then

begin
N ← 0;
Z ← (header + 4) EQL 0;  !queue empty
V ← tmp3 EQL 0;          !no entry to remove
C ← 0;
end;
else
begin
N ← 0;
Z ← 0;
V ← 1;                    !did not remove anything
C ← 1;                    !secondary interlock failed
end;

Exceptions: reserved operand

Opcodes:  5F   REMQTI Remove Entry from Queue at Tail, Interlocked

Description: The queue entry preceding the header is removed from the queue. The address operand is replaced by the address of the entry removed. If no entry was removed from the queue (because either there was nothing to remove or secondary interlock failed), the condition code V bit is set; otherwise it is cleared. If the interlock succeeded and the queue is empty at the end of this instruction, the condition code Z-bit is set; otherwise it is cleared. The removal is interlocked to prevent concurrent interlocked insertions or removals at the head or tail of the same queue by another process even in a multiprocessor environment. The removal is a 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. If the instruction fails to acquire the secondary interlock, the instruction sets condition codes and terminates without altering the queue.

Notes: 1. Because the removal is non-interruptible, processes running in kernel mode can share queues with interrupt service routines.
2. The INSQHI, INSQTI, REMQHI, and REMQTI instructions are implemented such that cooperating software processes in a multiprocessor may access a shared list without additional synchronization.

3. To release a software interlock realized with a queue, the following can be used:

\[\begin{align*}
1$: & REMQTI ... \quad ;\text{removed last?} \\
& \text{BEQL 2$} \quad ;\text{yes} \\
& \text{BCS 1$} \quad ;\text{try removing again} \\
& \text{CALL ACTIVATE(...)} \quad ;\text{Activate other waiters} \\
2$: & \\
\end{align*}\]

4. To remove entries until the queue is empty, the following can be used:

\[\begin{align*}
1$: & \text{REMQTI...} \quad ;\text{anything removed?} \\
& \text{BVS 2$} \quad ;\text{no} \\
& \quad \text{process removed entry} \\
& \quad \text{BR$1} \quad ; \\
2$: & \text{BCS$1} \quad ;\text{try removing again} \\
& \quad \text{queue empty} \\
\end{align*}\]

5. During access validation, any access which cannot be completed results in a memory management exception even though the queue removal is not started.

6. A reserved operand fault occurs if header, (header + 4), or (header + (header + 4)+4) is an address that is not quadword aligned (i.e., \(<2:0>\) NEQU 0) or if (header)<2: \(1\) is not zero. A reserved operand fault occurs if header equals addr. In this case the queue is not altered.
VARIABLE LENGTH BIT FIELD INSTRUCTIONS
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 (if the size is non-zero) 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 0 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 which the position is used to locate the bit field. The base address is obtained from an “address access” type operand. Unlike other “address access” type operands, register mode may be designated in the specifier. In this case, the field is contained in register n designated by the operand specifier (or register n+1 concatenated with register n).
Special Instructions

FIND FIRST
Purpose: locate first bit in bit field
Format: opcode startpos.rl, size.rb, base.vb, findpos.wl
Operation:

```
+----------------------------------+
| SIZE                            |
+----------------------------------+
| START POS                        |
```

SEARCH FOR 0 OR 1
RESULT IS FIND POSITIVE

Condition
N ← 0;
Z ← |bit not found|;
V ← 0;
C ← 0;

Codes:

Exceptions: Reserved operand

Opcodes:
EB    FFC    Find First Clear
EA    FFS    Find First Set

Description: A field specified by the start position, size, and base operands is extracted. The field is tested for a bit in the state indicated by the instruction starting at bit 0 and extending to the highest bit in the field. If a bit in the indicated state is found, the find position operand is replaced by the position of the bit and the Z condition code bit is cleared. If no bit in the indicated state is found, the find position operand is replaced by the position (relative to the base) of a bit one position to the left of the specified field and the Z condition code bit is set. If the size operand is 0, the find position operand is replaced by the start position operand and the Z condition code bit is set.

Notes: If start position + size is GEQU 231, 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, #20, Work, R3

;Find first bit
;set in work

Initial Conditions:
Work = ↑X 00040000 (Bit 18 set)
R3 = 00000000

197
Special Instructions

After Instruction Execution:
Work †X 00040000
R3 = 00000012hex (18 decimal)

Example:
FIND FIRST CLEAR
FFC #5, #10, Work1, R2

;Find first clear bit
;in Work1

Initial Conditions:
Work1 = †XF00
R2 = 00000000

After Instruction Execution:
Work1 = †XF00
R2 = 00000008

Example:
When referencing memory, the startpos field may be greater than 31. This provides an effective technique to search an entire array for the first bit set (or clear)

CLRL R0
10$:FFS R0,#32,ARRAY,R0

;start at bit 0
;R0 is incremented
;by 32 for each longword searched
;search next longword

BEQL10#

{R0 is match bit index}
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; 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.
Special Instructions

EXTRACT FIELD

Purpose: moves bit field to integer
Format: opcode pos.rl, size.rb, base.vb, dst.wl
Operation:

\[
\text{EXTV}
\]

\[
\text{EXTZV}
\]

Condition: \( N \leftarrow \text{dst} \text{ LSS } 0; \)

Codes:
- \( Z \leftarrow \text{dst EQL } 0; \)
- \( V \leftarrow 0; \)
- \( C \leftarrow C; \)

Exceptions: Reserved operand

Opcodes:
- EE \( \text{EXTV} \) Extract Field
- EF \( \text{EXTZV} \) 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.
Special Instructions

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 long-word 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, size NEQ 0, and the field is contained in the registers.
2. On a reserved operand fault, the destination operand is unaffected and the condition codes are unpredictable.

Example:

EXTRACT FIELD
EXTV #5, #10, Work1, R0 ;put bits 5 thru 14
;from Work1 into R0

Initial Conditions:
Work1 = 00004F04
R0 = 00000000

After Instruction Execution:
Work1 = 00004F04
R0 = FFFFFC78

Example:

EXTRACT FIELD, ZERO EXTENDED
EXTZV #5, #10, Work1, R1 ;put bits 5 thru 15
;from Work1 into R1
;and clear bits 11 thru 31

Initial Conditions:
Work1 = 00004F04
R1 = 00000000

After Instruction Execution:
Work1 = 00004F04
R1 = 00000478

200
**Special Instructions**

**COMPARE FIELD**

**Purpose:** compare bit field to integer  
**Format:** opcode pos.rl, size.rb, base.vb, src.rl  
**Operation:**

```
CMPV
```

```
CMPZV
```

**Condition Codes:**

- \( N \leftarrow \) field LSS src;
- \( Z \leftarrow \) field EQL src;
- \( V \leftarrow 0; \)
- \( C \leftarrow \) 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.

201
Special Instructions

Notes:

1. A reserved operand fault occurs if:
   a. size GTRU 32
   b. pos GTRU 31, size NEQ 0 and the field is contained in the registers.

2. On a reserved operand fault, the condition codes are unpredictable.

3. The comparison is with the entire source operand long-word, not just the size field.
Special Instructions

INSERT FIELD

Purpose: move integer to bit field
Format: opcode src.rl, pos.rl, size.rb, base.vb
Operation:

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

Exceptions: Reserved operand

Opcodes: F0 INSV Insert Field

Description: The field specified by the position, size, and base operands is replaced by bits size 1:0 of the source operand. If the size operand is 0, the only action is to affect the condition codes.

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

203
CHAPTER 8

CONTROL INSTRUCTIONS

INTRODUCTION
This chapter describes the branch, loop, control, subroutine, case, and call classes of instructions. In most implementations of the VAX-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.

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 instructions, and branch instructions take less space than jump instructions, the processor offers a variety of branch instructions to choose from. There are two unconditional branch instructions (branch and jump) and many conditional branch instructions.

The unconditional branch instructions allows you to specify a byte-size (BRB) or word-size displacement (BRW), which means you can branch to locations as far from the current location as 32,767 bytes in either direction. For control transfers to locations farther away, use the Jump instruction (JMP).

Two special types of branch and jump instruction are provided for calling subroutines: 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.
Control Instructions

B

BRANCH ON (CONDITION)

Purpose: test condition code
Format: opcode displ.bb
Operation: if condition then PC ← PC + SEXT (displ);
Condition
N ← N;
Z ← Z;
V ← V;
C ← C;

Codes: none

Opcodes:

<table>
<thead>
<tr>
<th>OPCODE</th>
<th>CONDITION</th>
<th>BNEQ, Branch on Not Equal (signed)</th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td>Z EQL 0</td>
<td>BNEQU Branch on Not Equal Unsigned</td>
</tr>
<tr>
<td>13</td>
<td>Z EQL 1</td>
<td>BEQL, Branch on Equal (signed)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>BEQU Branch on Equal Unsigned</td>
</tr>
<tr>
<td>14</td>
<td>[N OR Z] EQL 0</td>
<td>BGTR Branch on Greater Than (signed)</td>
</tr>
<tr>
<td>15</td>
<td>[N OR Z] EQL 1</td>
<td>BLEQ Branch on Less Than or Equal (signed)</td>
</tr>
<tr>
<td>18</td>
<td>N EQL 0</td>
<td>BGEQ Branch on Greater Than or Equal (signed)</td>
</tr>
<tr>
<td>19</td>
<td>N EQL 1</td>
<td>BLSS Branch on Less Than (signed)</td>
</tr>
<tr>
<td>1A</td>
<td>[C OR Z] EQL 0</td>
<td>BGTRU Branch on Greater Than Unsigned</td>
</tr>
<tr>
<td>1B</td>
<td>[C or Z] EQL 1</td>
<td>BLEQU Branch Less Than or Equal Unsigned</td>
</tr>
<tr>
<td>1C</td>
<td>V EQL 0</td>
<td>BVC Branch on Overflow Clear</td>
</tr>
<tr>
<td>1D</td>
<td>V EQL 1</td>
<td>BVS Branch on Overflow Set</td>
</tr>
<tr>
<td>1E</td>
<td>C EQL 0</td>
<td>BGEQU, Branch on Greater Than or Equal Unsigned</td>
</tr>
<tr>
<td></td>
<td></td>
<td>BCC Branch on Carry Clear</td>
</tr>
<tr>
<td>1F</td>
<td>C EQL 1</td>
<td>BLSSU, Branch on Less Than Unsigned</td>
</tr>
<tr>
<td></td>
<td></td>
<td>BSC Branch on Carry Set</td>
</tr>
</tbody>
</table>

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.
Control Instructions

Notes:
The VAX-11 conditional branch instructions permit considerable flexibility in branching but require care in choosing 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
   BEQU      Z EQL 1
   BNEQU     Z EQL 0
   BGEQU     C EQL 0
   BGTRU     |C OR Z| EQL 0

   These instructions typically follow integer and field instructions where the operands are treated as unsigned integers, addressed instructions, and character string instructions.

3. Signed Group
   BLSS       N EQL 1
   BLEQ       |N OR Z| EQL 1
   BEQ        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.
Control Instructions

BR
JMP

Purpose: transfer control
Format: opcode displ.bx
        opcode dst.ab
Operation: PC ← PC + SEXT (displ);
          PC ← dst;
Condition Codes:
                N ← N;
                Z ← Z;
                V ← V;
                C ← C;
Exceptions: none
Opcodes:
            11 BRB Branch With Byte Displacement
            31 BRW Branch With Word Displacement
            17 JMP Jump
Description: For branch, the sign-extended branch displacement is added to PC and PC is replaced by the result. For Jump, the PC is replaced by the destination operand.
Control Instructions

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: N ← N;
Codes: 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.
3. The modification of the bit is not an interlocked operation. See BBSSII and BBCCI for interlocking instructions.
Control Instructions

BB

BRANCH ON BIT (AND MODIFY WITHOUT INTERLOCKED)

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: N ← N;
Codes: 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 BTRU 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] the 1 else 0;
newstate = teststate;
{set interlock};
temp ← FIELD (pos, 1, base);
FIELD (pos, 1, base) ← newstate;
{release interlock};
if tmp EQL teststate then
PC ← PC + SEXT (displ);

Condition
N ← N;

Codes:
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. The operation is interlocked against similar operations by other processors or devices in the system.

Notes: 1. A reserved operand fault occurs if pos GTRU 31 and the bit is contained in registers.
2. On a reserved operand fault, the field is unaffected and the condition codes are unpredictable.
3. Except for memory interlocking BBSSI is equivalent to BBSS and BBCCI is equivalent to BBCC.

Example: This instruction is designed to implement interlock with other processors or devices. For example, to implement "busy waiting":

1$ BBSSI bit,base,1$
BLB

Purpose: test bit
Format: opcode src.rl, displ.bb
Operation: teststate = if {BLBS} then 1 else 0;
if src<0> EQL teststate then
PC ← PC + SEXT (displ);
Condition: N ← N;
Codes: 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 re-
placed by the result.
Notes: The source operand is taken with longword context although
only one bit is tested.
LOOP CONTROL INSTRUCTIONS
Control Instructions

ACB

ADD COMPARE AND BRANCH

Purpose: maintain loop count and loop
Format: opcode limit.rx, add.rx, index.mx, displ.bw
Operation: index ← index + add;
if ( | add GEQ 0 | AND | index LEQ limit | | OR |
| | add LSS 0 | AND | index GEQ limit | ) then
PC ← PC + SEXT (displ);
Condition
N ← index LSS 0;
Codes:
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 normal-
ly.
4. On floating overflow, the index operand is replaced by an
operand of all bits 0 except for a sign bit of 1 (reserved
operand). N ← 1; Z ← 0; V ← 1. The branch is not taken.

214
Control Instructions

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.
AOB

ADD ONE AND BRANCH

Purpose: increment integer loop count and loop

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

Operation:
index ← index 1;
if index LSS limit then PC ← PC + SEXT (displ);
if index LEQ limit then PC ← PC + SEXT (displ);

!AOBLSS

Condition N ← index LSS 0;

Codes: Z ← index equal 0;
C ← {integer overflow};
C ← C;

Exceptions: integer overflow

Opodes: F2 AOBLLS 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 AOBLLS, if it is less than, the branch is taken. On AOBLEQ, if it is less than or equal, the branch is taken. If the branch is taken, the sign extended branch displacement is added to the PC and the PC is replaced by the result.

Notes:
1. Integer overflow occurs if the index operand before addition is the largest positive integer. On overflow, the index operand is replaced by the largest negative integer, and thus (unless the limit operand is the largest negative integer on AOBLLS) the branch is taken.
2. The C-bit is unaffected.
CONTROL INSTRUCTIONS

SUBTRACT ONE AND BRANCH

Purpose: decrement integer loop count and loop

Format: opcode index.ml, displ.bb

Operation:
index ← index − 1;
If index GEO 0 then !SOBGEQ
PC ← PC + SEXT (displ);
index ← index−1:
If index GTR 0 then !SOBGTR
PC ← PC + SEXT (displ);

Condition:
N ← index LSS 0;
Z ← index EQL 0;
V ← [integer overflow];
C ← C;

Exceptions: integer overflow

Opcodes:
F4 SOBGEQ Subtract One and Branch Greater Than or Equal
F5 SOBGTR Subtract One and Branch Greater Than

Description: One is subtracted from the index operand and the index operand is replaced by the result. On SOBGEQ, if the index operand is greater than or equal to 0, the branch is taken. On SOBGTR, if the index operand is greater than 0, the branch is taken. If the branch is taken, the sign-extended branch displacement is added to the PC and the PC is replaced by the result.

Notes:
1. Integer overflow occurs if the index operand before subtraction is the largest negative integer. On overflow, the index operand is replaced by the largest positive integer, and thus the branch is taken.
2. The C-bit is unaffected.
CASE

CASE INSTRUCTIONS

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 tmp LEQU limit then
SEXT (displ [tmp]) else [2 + 2* ZEST (limit)];

Condition:
N ← temp LSS limit;

Codes:
Z ← temp EQL limit;
V ← 0;
C ← temp 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 com-
pared with the limit operand and if it is less than or equal
unsigned, a branch displacement selected by the temporary
value is added to PC and PC is replaced by the result.
Otherwise, 2 times the sum of the limit operand plus 1 is added
to PC and PC is replaced by the result. This causes PC to be
moved past the array of branch displacements. Regardless of
the branch taken, the condition codes are affected by the com-
parison 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 rela-
tive to the address of displ [0].
2. The selector and base operands can both be considered
either as signed or unsigned integers.
3. The limit is |the number of choices|−1.

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 de-
pending on the value you obtain as a selector. The branch falls
through if the selector does not generate any of the displace-
ments on the list.
Control Instructions

The FORTRAN STATEMENT
GO TO (10, 20, 30), I
is equivalent to

CASEL I, #1, #3 ; only values 1,2,3 are valid
1$ .WORD 10$−1$, #3 ; if 1
 .WORD 20$−1$ ; if 2
 .WORD 20$−1$ ; if 3
 ; fall through if out of range

10$:

20$:

30$:
Control Instructions

BSB
JSB

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);
PC ← dst;

Condition: N ← N;
Codes: 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: \( \text{PC} \leftarrow (\text{SP}) + ; \)
Condition: \( \text{N} \leftarrow \text{N}; \)
Codes: \( \text{Z} \leftarrow \text{Z}; \)
\( \text{V} \leftarrow \text{V}; \)
\( \text{C} \leftarrow \text{C}; \)
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 one byte shorter.
PROCEDURE CALL INSTRUCTIONS
Procedures are general purpose routines that use argument lists passed automatically by the processor and use only local variables for data storage. 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:

```
  15 14 13 12 11
     DV IV MBZ         REGISTERS
```

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 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 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 save 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:

<table>
<thead>
<tr>
<th>SPA</th>
<th>S</th>
<th>D</th>
<th>MASK</th>
<th>PSW</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>SAVED AP</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>SAVED FP</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>SAVED PC</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>SAVED R0</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>SAVED R11</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(0 TO 3 BYTES SPECIFIED BY SPA)
$ S$ BIT = SET IF CALLS; CLEAR IF CALLG.

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:

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>SF$A_HANDLER</td>
<td>0</td>
<td>condition handler</td>
</tr>
<tr>
<td>SF$W_SAVE_PSW</td>
<td>4</td>
<td>saved PSW</td>
</tr>
<tr>
<td>SF$W_SAVE_MASK</td>
<td>6</td>
<td>offset,CALLS, and mask</td>
</tr>
<tr>
<td>SF$L_SAVE_AP</td>
<td>8</td>
<td>saved AP</td>
</tr>
<tr>
<td>SF$L_SAVE_FP</td>
<td>12</td>
<td>saved FP (backward link)</td>
</tr>
<tr>
<td>SF$L_SAVE_PC</td>
<td>16</td>
<td>saved PC</td>
</tr>
<tr>
<td>SF$L_SAVE_REGS</td>
<td>20</td>
<td>start of saved R0..RT11</td>
</tr>
</tbody>
</table>
Control Instructions

The savepsw fields have symbolic names as follows:

SF$a_C  <0>  saved C condition code
SF$a_V  <1>  saved V condition code
SF$a_Z  <2>  saved Z condition code
SF$a_N  <3>  saved N condition code
SF$a_TBIT <4>  saved trace enable
SF$a_IV  <5>  saved integer overflow enable
SF$a FU  <6>  saved floating underflow enable
SF$a DV  <7>  saved divide overflow enable

The save_mask field have symbolic names as follows:

SF$b_SAVE_MASK   <11:0>  register mask
SF$b_CALLS       <13>  CALLS flag
SF$b_STACKOFFS   <15:14> stack alignment

a = M,S, or V for mask, size, or position
b = S or V for size or position

These names are defined by the $SFDEF macro in the system library.
PROCEDURE CALL INSTRUCTIONS
CALLG

CALL PROCEDURE WITH GENERAL ARGUMENT LIST

Purpose: invoke a procedure with actual arguments from anywhere in memory

Format: opcode arglist.ab, dst.ab

Operation: 
- [align stack];
- [create stack frame];
- [set arithmetic trap enables];
- [set new values of AP, FP, PC];

Condition: N ← 0;

Codes: 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 which specifies the address of the actual argument list. The trap enables in the PSW are set to a known state. Integer overflow, and decimal overflow are affected according to bits 14 and 15 of the entry mask respectively; floating underflow is cleared. T-bit is unaffected. PC is replaced by the sum of destination operand plus 2 which transfers control to the called procedure at the byte beyond the entry mask.
Control Instructions

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.

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.)
CALLS

CALL PROCEDURE WITH STACK ARGUMENT LIST

Purpose: invoke a procedure with actual arguments or addresses on the stack

Format: opcode numarg.rl,dst.ab

Operation: [push arg count];
[align stack];
[create stack frame];
[set arithmetic trap enables];
[set new values of AP, FP, PC];

Condition
Codes:

N ← 0;
Z ← 0;
V ← 0;
C ← 0;

Exceptions: reserved operand

Opcodes: FB CALLS Call Procedure With Stack Argument List

Description: The numarg operand is pushed on the stack as a longword (byte 0 contains the number of arguments high order 24 bits are used by DIGITAL software). SP is saved in a temporary and then bits 1:0 of SP are replaced by 0 so that the stack is longword aligned. The procedure entry mask is scanned from bit 11 to bit 0 and the contents of register whose number corresponds to set bits in the mask are pushed on the stack. PC, FP, and AP are pushed on the stack as longwords. The condition codes are cleared. A longword containing the saved two low bits of SP in bits 31:30, a 1 in bit 29, a 0 in bit 28, the low 12 bits of the procedure entry mask in bits 27:16, and the PSW in bits 15:0 with T cleared is pushed on the stack. A longword 0 is pushed on the stack. FP is replaced by SP. AP is set to the saved SP (the value of the stack pointer after the number of arguments operand was pushed on the stack). The trap enables in the PSW are set to a known state. Integer overflow and decimal overflow are affected according to bits 14 and 15 of the entry mask, respectively; floating underflow is cleared. 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 and R11 which are modified in the called procedure must be preserved in the entry mask. (Refer to Appendix C.)
RET

RETURN FROM PROCEDURE

Purpose: transfer control from a procedure back to calling program

Format: opcode

Operation: [restore SP from FP];
          [restore registers];
          [drop stack alignment];
          [restore PSW];
          [If CALLS, remove arglist];

Condition Codes:

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 in a temporary PC, FP, and AP are replaced by long- words 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 temporary1<15:8> NEG 0.
2. On a reserved operand fault, the condition codes are unpredictable. The value of temporary1<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

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

CHARACTER STRING INSTRUCTIONS
A character string is specified by 2 operands:
1. An unsigned word operand which specifies the length of the char-
   acter 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 execu-
tion of the instruction. At completion, these registers are available to
software to use as string specification operands for a subsequent
instruction. During the execution of the instructions, pending interrupt
conditions are tested and if any are found, the control block is updat-
ed, a first part done bit is set in the PSL, and the instruction interrupted
(refer to Chapter 12, EXCEPTIONS, for greater detail). After the inter-
ruption, the instruction resumes transparently. The format of the
control block is:

<table>
<thead>
<tr>
<th>ADDRESS 1</th>
<th>LENGTH 1</th>
<th>R0</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADDRESS 2</td>
<td>LENGTH 2</td>
<td>R1</td>
</tr>
<tr>
<td>ADDRESS 3</td>
<td>LENGTH 3</td>
<td>R2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>R3</td>
</tr>
<tr>
<td></td>
<td></td>
<td>R4</td>
</tr>
<tr>
<td></td>
<td></td>
<td>R5</td>
</tr>
</tbody>
</table>

The fields LENGTH 1, LENGTH 2 (if required) and LENGTH 3 (if re-
quired) contain the number of bytes remaining to be processed in the
first, second and third string operands respectively. The fields AD-
DRESS 1, ADDRESS 2 (if required) and ADDRESS 3 (if required) con-
tain 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 associ-
ated with the instruction descriptions.

233
**MOVC**

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

**Format:**
- op addr.ab, dst addr.ab
- op addr.ab, src len, dst len, dst addr.ab

**Operation:**

**MOVC3, MOVC5** if src len = dst len

```plaintext
MOVC3, MOVC5 if src len = dst len
```

**MOVC5** if src len > dst len

```plaintext
MOVC5 if src len > dst len
```

**MOVC5** if src len < dst len

```plaintext
MOVC5 if src len < dst len
```

**Condition Codes:**
- N ← src len LSS dst len;
- Z ← src len EQL dst len;
- V ← 0;
- C ← src len LSSU dst len;

**MOVE CHARACTER**

3 operand

5 operand

\[ C = 0, Z = 1 \]

\[ C = 0, Z = 0 \]

\[ C = 1, Z = 0 \]
Character String Instructions

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 address 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.

   5. On MOVC3, or if the MOVC5 and the strings are of equal length, then Z is set and N, V, and C are cleared.
MOVTC

MOVE TRANSLATED CHARACTERS

Purpose: to move and translate character string
Format: opcode src.len.rw, src.addr.ab, fill.rb, tbl.addr.ab, dst.len.rw, dst.addr.ab
Operation:

\[
\text{MOVTC} \quad \text{src} \longrightarrow \text{dst} \quad \text{len}
\]

NOTE: THE CASE OF src.len = dst.len AND src.len > dst.len
SIMILAR TO THAT SHOWN IN THE MOVCS INSTRUCTION

Condition
Codes:
N ← src.len LSS dst.len;
Z ← src.len EQL dst.len;
V ← 0;
C ← src.len LSSU dst.len;

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.
Character String Instructions

Notes:

1. 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 on one byte beyond the destination string.
MOVTUC

MOVE TRANSLATED UNTIL CHARACTER

Purpose: to move and translate character string, handling escape codes

Format: opcode src.len rw, src.add ab, esc.rb, tbl.add ab, dst.len rw, dst.add ab

Operation:

```
STOP IF OUTPUT = esc
NO FILL CHARACTERS
V SET IF esc
Z SET IF SAME SIZE
C SET IF src.len < dst
```

Condition

Codes: 

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 and R0 through R5 are unpredictable.
Character String Instructions

Notes:

1. After execution:

   R0 = number of bytes remaining in source string (including the byte which caused the escape). R0 is zero only if the entire source string was translated and moved without escape.

   R1 = address of the byte which resulted in destination string exhaustion or escape; or if no exhaustion or escape, R1 = address of one byte beyond the source string.

   R2 = 0

   R3 = address of the table.

   R4 = number of bytes remaining in the destination string.

   R5 = address of the byte in the destination string which would have received the translated byte that caused the escape or would have received a translated byte if the source string were not exhausted; or if no exhaustion or escape, R1 = address of one byte beyond the destination string.

2. V should be tested before the V and C condition codes to make sure that an escape is detected on the last character of the source string.

For example:

10$: MOVTUC R0,(R1),# escape,table,R4,(R5)

   BVS HANDLE_ESCAPE
   BGTRU DST_TOO_SMALL

   [OK]

   [OK]

   HANDLE_ESCAPE:

   [update R0,R1,R4,R5 for escape handling]

   [continue translation]

   BRB 10$ ;continue translation

239
Character String Instructions

CMPC

Purpose: to compare two character strings
Format: opcode len.rw,
src1addr.ab, src2addr.ab
opcode src1len.rw,
src1addr.ab, fill.rb
src2len.rw, src2addr.ab

Operation:

Condition Codes: !Final Condition codes reflect last affecting
!of Condition Codes in Operation above.
N ← \([first\ byte]\ LSS [second\ byte]\);
Z ← \([first\ byte]\ EQL [second\ byte]\);
V ← 0;
C ← \([first\ byte]\ LSSU [second\ byte]\);

Exceptions: None

Opcodes: 29 CMPC3 Compare Characters 3 Operand
2D CMPC5 Compare Characters 5 Operand

Description: In 3 operand format, the bytes of string 1 specified by the
length and address 1 operands are compared with the bytes of
string 2 specified by the length and address 2 operands. Compar-
ison 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
Character String Instructions

the length of the longer by appending (at higher addresses) bytes equal to the fill operand. Comparison proceeds until inequality is detected or all the bytes of the strings have been examined. Condition codes are affected by the result of the last byte comparison.

Notes:

1. After execution of CMPC3:

   R0 = number of bytes remaining in string 1 (including byte which terminated comparison); R0 is zero only if strings are equal.

   R1 = address of the byte in string 1 which terminated comparison; if strings are equal, R1 = address of one byte beyond string 1.

   R2 = R0

   R3 = address of the byte in string 2 which terminated comparison: if strings are equal, R3 = address of one byte beyond string 2.

2. After execution of CMPC5:

   R0 = number of bytes remaining in string 1 (including byte which terminated comparison); R0 is zero only if string 1 and string 2 are of equal length and equal or string 1 was exhausted before comparison terminated.

   R1 = address of the byte in string 1 which terminated comparison; if comparison did not terminate before string 1 exhausted, R1 = address of one byte beyond string 1.

   R2 = number of bytes remaining in string 2 (including byte which terminated comparison); R0 is zero only if string 2 and string 1 are of equal length or string 2 was exhausted before comparison terminated.

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

3. If both strings have zero length, Z is set and N, V and C are cleared just as in the case of two equal strings.
SCAN CHARACTERS, SPAN CHARACTERS

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

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

**Operation:**

```
SCANC
SPANC
```

```
<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>addr</td>
<td>len</td>
</tr>
</tbody>
</table>

MASK TEST EACH CHARACTER UNTIL ZERO (SPANC) OR NOT ZERO (SCANC)

```

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>tbl</td>
<td>addr</td>
</tr>
</tbody>
</table>

256

**Condition**

\[ N \leftarrow 0; \]

\[ Z \leftarrow R0 \text{ EQL } 0; \]

\[ V \leftarrow 0; \]

\[ C \leftarrow 0; \]

**Codes:**

Z SET IF CONDITION NOT SATISIFIED

**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 ANDef 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:**

1. After execution:

   \[ R0 = \text{number of bytes remaining in the string (including the byte which produced the non-zero AND result for SCANC or zero result for SPANC).} \]

   \[ R0 \text{ is zero only if there was a zero AND result for SCANC or a non-zero result for SPANC.} \]

   \[ R1 = \text{address of the byte which produced non-zero AND result for SCANC or a zero AND result for SPANC; or, if zero result, } R1 = \text{address of one byte beyond the string.} \]

   \[ R2 = 0 \]

   \[ R3 = \text{address of the table} \]

2. If the string has zero length, condition code Z is set just as though the entire string were scanned (spanned).
LOCATE CHARACTER, SKIP CHARACTER

Purpose: to find or skip character in character string
Format: opcode char.rb, len.rw, addr.ab
Operation:

\[ \text{LOCC, SKPC} \]

COMPARE EACH CHARACTER UNTIL EQUAL (LOCC) OR NOT EQUAL (SKPC)

Z SET IF CONDITION NOT SATISFIED

Condition: \( N \leftarrow 0; \)
Codes: \( Z \leftarrow R0 \text{ EQL 0}; \)
\( V \leftarrow 0; \)
\( C \leftarrow 0; \)

Exceptions: None

Opcodes: 3A \text{ LOCC} \quad \text{Locate Character}
3B \text{ SKPC} \quad \text{Skip Character}

Description: The character operand is compared with the bytes of the string specified by the length and address operands. Comparison continues until equality is detected for the Locate Character instruction or inequality for the Skip Character instruction or until all bytes of the string have been compared. If equality is detected for the Locate Character instruction, the condition code Z bit is cleared; otherwise the Z bit is set. If inequality is detected for the Skip Character instruction, the condition code Z bit is cleared; otherwise the Z bit is set.

Notes:
1. After execution:
   \( R0 = \text{number of bytes remaining in the string (including located one) if byte located; otherwise } R0 = 0. \)
   \( R1 = \text{address of the byte located if byte located; other } R1 = \text{ address of one byte beyond the string.} \)
2. If the string has zero length, condition code Z is set just as though each byte of the entire string were equal (unequal) to the character.
**MATCHC**

**Purpose:** to find substring (object) in character string

**Format:** opcode objlen.rw, objaddr.ab, srclen.rw, srcaddr.ab

**Operation:**

```
MATCHC

SEARCH

FULL STRING MATCH

srcaddr

srclen

objlen

objaddr

object
```

**Condition Codes:**

- \( N \leftarrow 0; \)
- \( Z \leftarrow R0 \ EQL \ 0; \)  if match found
- \( V \leftarrow 0; \)
- \( C \leftarrow 0; \)

**Exceptions:** None

**Opcodes:** 39 MATCHC Match Characters

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

**Notes:**

1. After execution:
   - \( R0 = \) if a match occurred 0; otherwise the number of bytes in the object string.
   - \( R1 = \) if a match occurred, the address of one byte beyond the object string; otherwise the address of the object string.
   - \( R2 = \) if a match occurred, the number of bytes remaining in the source string after the match; otherwise 0.
   - \( R3 = \) if a match occurred, the address of 1 byte beyond the last byte matched; otherwise the address of 1 byte beyond the source string.

2. If both strings have zero length or if the object string has zero length, condition code Z is set just as though the substring were found.

3. If the source string has zero length and the object string has non-zero length, condition code Z is cleared just as though the substring were not found.
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 “Cyclic 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 non-contiguous strings.

The CRC instruction operates by scanning the string, and for each byte of the data stream, including it in the CRC being calculated. The byte is included by XORing it to the right eight bits of the CRC. Then the CRC is shifted right one bit, inserting zero on the left. The 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.
CRC

CALCULATE CYCLIC REDUNDANCY CHECK

Purpose: communications or software redundancy checks
Format: opcode tbl.ab, inicrc_rl, strlen_rw, stream.ab,
Operation:

Condition
Codes:
None
Opcodes: 0B CRC Calculate Cyclic Redundancy Check
Description: The CRC of the data stream described by the string descriptor is calculated. The initial CRC is given by inicrc and is normally 0 or −1 unless the CRC is calculated in several steps. R0 is replaced by the result. If the polynomial is less than order 32, the result must be extracted from R0. The CRC polynomial is expressed by the contents of the 16-longword table. See the notes for calculation of the table.

Notes:
1. If the data stream is not a multiple of eight bits long, it must be right adjusted with leading zero fill.
2. If the CRC polynomial is less than order 32, the result must be extracted from the low order bits of R0.
3. The following algorithm can be used to calculate the CRC table given a polynomial expressed as follows:
   \[ \text{poly}_n \leftarrow |\text{coefficient of } x^{*n}| \]

  This routine is available as system library routine LIB$CRC_TABLE (poly.rl, table.ab). The table is the loca-
Character String Instructions

ition of a 64-byte (16-longword) table into which the result will be written.

SUBROUTINE LIB$CRC_TABLE (POLY, TABLE)
INTEGER*4 POLY, TABLE(0:15), TMP, X
DO 190 INDEX = 0, 15
  TMP = INDEX
  DO 150 I = 1,4
    X = TMP .AND. 1
    TMP = 1SHFT (TMP, -1) !logical shift right one bit
  IF (X .EQ.1) TMP = TMP .XOR. POLY
  150 CONTINUE
  TABLE(INDEX) = TMP
190 CONTINUE
RETURN
END

4. The following are descriptions of some commonly used CRC polynomials.

  CRC-16 (used in DDCMP and Bisync)
  polynomial: \( x^{16} + x^{15} + x^2 + 1 \)
  poly: 120001 (octal)
  initialize: 0
  result: \( R0 <15:0> \)

  CCITT (used in ADCCP, HDLC, SDLC)
  polynomial: \( x^{16} + x^{12} + x^5 + 1 \)
  poly: 102010 (octal)
  initialize: \(-1<15:0>\)
  result: complement of \( R0<15:0> \)

  AUTODIN-II
  polynomial \( x^{32} + x^{26} + x^{23} + x^{22} + x^{16} + x^{12} +x^{11} + x^{10} + x^{8} + x^{7} + x^{5} + x^{4} + x^{2} + x + 1 \)
  poly: EDB88320 (hex)
  initialize: \(-1<31:0>\)
  result: complement of \( R0<31:0> \)

5. This instruction produces an unpredictable result unless the table is well formed, such as produced in note 3. Note that for any well formed table, entry [0] is always 0 and entry [8] is always the polynomial expressed as in note 3. The operation can be implemented using shifts of one, two, or four bits at a time as follows:
### Character String Instructions

<table>
<thead>
<tr>
<th>steps per byte</th>
<th>table index</th>
<th>table multiplier</th>
<th>use table entries</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>tmp3&lt;0&gt;</td>
<td>8</td>
<td>[0] = 0, &lt;8&gt;</td>
</tr>
<tr>
<td>2</td>
<td>tmp3&lt;1:0&gt;</td>
<td>4</td>
<td>[0] = 0, [4], [8], [12]</td>
</tr>
<tr>
<td>4</td>
<td>tmp3&lt;3:0&gt;</td>
<td>1</td>
<td>all</td>
</tr>
</tbody>
</table>

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

INTRODUCTION

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 phase "decimal string" is used, it means any of the three data types.

A decimal string is specified by two operands:

1. For decimal strings, the length is the number of digits in the string. The number of bytes in the string is a function of the length and the type of decimal string referenced.

2. The address of the lowest addressed byte of the string. This byte contains the most significant digit for Trailing Numeric and Packed Decimal strings. This byte contains a sign for leading Separate Numeric strings. The address is specified by a byte operand of address access type.

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 is interrupted. After the interruption, the instruction resumes transparently. The format of the control block at completion is:

```
<table>
<thead>
<tr>
<th>31</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>ADDRESS 1</td>
<td>R0</td>
</tr>
<tr>
<td>0</td>
<td>R1</td>
</tr>
<tr>
<td>ADDRESS 2</td>
<td>R2</td>
</tr>
<tr>
<td>0</td>
<td>R3</td>
</tr>
<tr>
<td>ADDRESS 3</td>
<td>R4</td>
</tr>
<tr>
<td>0</td>
<td>R5</td>
</tr>
</tbody>
</table>
```
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.

**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.

**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.

**RESERVED OPERAND EXCEPTION**
A reserved operand fault occurs if the length of a decimal string operand is outside the range 0 through 31, or if an invalid sign or digit is encountered in CVTSP and CVTTP.

**UNPREDICTABLE RESULTS**
The result of any operation is unpredictable if any source decimal string operand contains invalid data. Except for CVTSP and CVTTP, the decimal string instructions do not verify the validity of source operand data.

If the destination operands overlap any source operands, the result of an operation will, in general, be unpredictable. The destination strings, registers used by the instruction, and condition codes will, in general, be unpredictable when a reserved operand fault occurs.

**PACKED DECIMAL OPERATIONS**
Packed Decimal strings generated by the decimal string instructions always have the preferred sign representation: 12 for “+” and 13 for
Decimal String Instructions

"-". 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.

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.
**MOVP**

**MOVE PACKED**

**Purpose:** move a packed decimal string from one memory location to another

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

**Operation:** |dst string| ← |src string|;

**Condition**

N ← |dst string| LSS 0;

Z ← |dst string| EQL 0;

V ← 0;

C ← C;

**Exceptions:** reserved operand

**Opcodes:** 34 MOV P 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

**Purpose:** compare two packed decimal strings and set condition codes

**Format:**
- opcode len.rw, scr1addr.ab, src2addr.ab  
  3 operand
- opcode src1len.rw, src1addr.ab, src2len.rw, src2addr.ab  
  4 operand

**Operation:** \([\text{src1 string}] \rightarrow [\text{src2 string}]\);

**Condition:**\( N \leftarrow [\text{src1 string}] \text{ LSS} [\text{src2 string}]\);

**Codes:**
- \( Z \leftarrow [\text{src1 string}] \text{ EQL} [\text{src2 string}]\);
- \( V \leftarrow 0; \)
- \( C \leftarrow 0; \)

**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:
   - \( R0 = 0 \)
   - \( R1 = \text{address of the byte containing the most significant digit of string 1.} \)
   - \( R2 = 0 \)
   - \( R3 = \text{address of the byte containing the most significant digit of string 2.} \)
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**

**Purpose:** add one packed decimal string to another

**Format:**
- opcode addlen.rw, addaddr.ab, sumlen.rw, sumaddr.ab
- opcode add1len.rw, add1addr.ab, add2len.rw, add2addr.ab, sumlen.rw, sumaddr.ab

**Operation:**
- \[\text{sum string} \leftarrow \text{sum string} + \text{add string};\]
- \[\text{sum string} \leftarrow \text{add 1 string} + \text{add 2 string};\]

**Condition:**
- \(N \leftarrow \text{sum string} \text{ LESS 0};\)
- \(Z \leftarrow \text{sum string} \text{ EQL 0};\)
- \(V \leftarrow \text{decimal overflow};\)
- \(C \leftarrow 0;\)

**Exceptions:**
- reserved operand
- decimal overflow

**Opcodes:**
- 20 ADDP4 Add Packed 4 Operand
- 21 ADDP6 Add Packed 6 Operand

**Description:**
In 4 operand format, the addend string specified by the addend length and addend address operands is added to the sum string specified by the sum length and sum address operands and the sum string is replaced by the result.

In 6 operand format, the addend 1 string specified by the addend 1 length and addend 1 address operands is added to the addend 2 string specified by the addend 2 length and addend 2 address operands. The sum string specified by the sum length and sum address operands is replaced by the result.

**Notes:**
1. After execution of ADDP4:
   - \(R = 0\)
   - \(R1 = \text{address of the byte containing the most significant digit of the addend string}\)
   - \(R2 = 0\)
   - \(R3 = \text{address of the byte containing the most significant digit of the sum string}\)
2. After execution of ADDP6:
   - \(R0 = 0\)
   - \(R1 = \text{address of the byte containing the most significant digit of the addend1 string}\)
   - \(R2 = 0\)
Decimal String Instructions

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 falt occurs.

4. If all destination digits are zero, Z is set and N is cleared. This is true even if the result overflows.
Decimal String Instructions

**SUBP**

**SUBTRACT PACKED**

**Purpose:** subtract one packed decimal string from another

**Format:**
- opcode sublen.rw, subaddr.ab, diflen.rw, difaddr.ab
- 4 operand

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

**Operation:**
- |dif string| ← |dif string| − |sub string|;
- 4 operand

- |dif string| ← |min string| − |sub string|;
- 6 operand

**Condition Codes:**
- \( N \leftarrow |\text{dif string}| \text{ LSS} 0; \)
- \( Z \leftarrow |\text{dif string}| \text{ EQL} 0; \)
- \( V \leftarrow |\text{decimal overflow}|; \)
- \( C \leftarrow 0; \)

**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:
   - \( R0 = 0 \)
   - \( R1 = \text{address of the byte containing the most significant digit of the subtrahend string} \)
   - \( R2 = 0 \)
   - \( R3 = \text{address of the byte containing the most significant digit of the difference string} \)

2. After execution of SUBP6:
   - \( R0 = 0 \)
   - \( R1 = \text{address of the byte containing the most significant digit of the subtrahend string} \)
   - \( R2 = 0 \)
Decimal String Instructions

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.
MULP

MULTIPLY PACKED

Purpose: multiply one packed decimal string by a second, result placed in a third

Format: opcode mulrlen.rw, mulraddr.ab, muldlen.rw, muldaddr.ab, prodlen.rw, prodaddr.ab

Operation: |prod string| ← |muld string| * |mulr string|;

Condition: N ← |prod string| LSS 0;

Codes: 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

Purpose: divide one packed decimal string by a second, result placed in a third

Format: opcode divrlen.rw, divraddr.ab, divdlen.rw, divdaddr.ab, quo.len.rw, Quoaddr.ab

Operation: \(|\text{quo string}\) \(\leftarrow\) \(|\text{divd string}\) / \(|\text{divr string}\);

Condition: \(N \leftarrow |\text{quo string}| \) LSS 0;
Codes: \(Z \leftarrow |\text{quo string}| \) EQL 0;
\(V \leftarrow \) decimal overflow;
\(C \leftarrow 0;\)

Exceptions: reserved operand
decimal overflow
divide by zero

Opcodes: 27 DIVP

Description: The dividend string specified by the dividend length and dividend address operands is divided by the divisor string specified by the divisor length and divisor address operands. The quotient string specified by the quotient length and quotient address operands is replaced by the result.

Notes:

1. This instruction may allocate a 16 byte workspace on the stack. After execution SP is restored to its original contents and the contents of \(|(SP) - 16| \) * \(|(SP) - 1|\) are unpredictable.

2. The division is performed such that:
   - The absolute value of the remainder (which is lost) is less than the absolute value of the divisor.
   - The product of the absolute value of the quotient times the absolute value of the divisor is less than or equal to the absolute value of the dividend.
   - The sign of the quotient is determined by the rules of algebra from the signs of the dividend and the divisor. If the value of the quotient is zero, the sign is always positive.

3. After execution:
   \(R0 = 0\)
   \(R1 = \) address of the byte containing the most significant digit of the divisor string
   \(R2 = 0\)
   \(R3 = \) address of the byte containing the most significant digit of the dividend string
Decimal String Instructions

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

Purpose: convert longword integer to packed decimal string

Format: opcode src.rl, dstlen.rw, dstaddr.ab

Operation: \( \text{[dst string]} \leftarrow \text{conversion of src}; \)

Condition: \( \text{N} \leftarrow \text{[dst string]} \text{ LSS 0}; \)

Codes: \( \text{Z} \leftarrow \text{[dst string]} \text{ EQL 0}; \)
\( \text{V} \leftarrow \text{[decimal overflow]}; \)
\( \text{c} \leftarrow 0; \)

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:
   \( \text{R0} = 0 \)
   \( \text{R1} = 0 \)
   \( \text{R2} = 0 \)
   \( \text{R3} = \text{address of the byte containing the most significant digit of the destination string} \)
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.
CVTPL

CONVERT PACKED TO LONG

Purpose: convert packed decimal string to longword integer
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 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 conver-
     sion. 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

Purpose: convert packed decimal string to trailing numeric string

Format: opcode srclen.rw, srcaddr.ab, tbladdr.ab, dstlen.rw, dstaddr.ab

Operation: \[ \text{dst string} \leftarrow \text{conversion of src string}; \]

Condition: \[ N \leftarrow \text{src string}; \text{LSS 0}; \]

Codes: \[ Z \leftarrow \text{src string}; \text{EQL 0}; \]

V \leftarrow \text{decimal overflow}; \]

C \leftarrow 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:
   \[ R0 = 0 \]
   \[ R1 = \text{address of the byte containing the most significant digit of the source string} \]
   \[ R2 = 0 \]
   \[ R3 = \text{address of the most significant digit of the destination string} \]

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 negative 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).
Decimal String Instructions

CONVERT TRAILING NUMERIC TO PACKED

Purpose: convert trailing numeric string to packed decimal string

Format: opcode srclen.rw, srcaddr.ab, tbladdr.ab,
dstlen.rw, dstaddr.ab

Operation: |dst string| ← conversion of |src string|;

Condition
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 (trail-
ing) 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 contain-
ing 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:
   • The length of the source trailing numeric string is
     outside the range 0 through 31.
   • The length of the destination packed decimal string
     is outside the range 0 through 31.
   • The source string contains an invalid byte. An invalid
     byte is any value other than ASCII “O” through “9” in
     any high order byte (i.e., any byte except the least
     significant byte).
   • The translation of the least significant digit produces
     an invalid packed decimal digit or sign nibble.

2. After execution:
   R0 = 0
   R1 = address of the most significant digit of the source
       string
   R2 = 0

267
Decimal String Instructions

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

Purpose: convert packed decimal string to leading separate numeric string

Format: opcode srclen.rw, srcaddr.ab, dstlen.rw, dstaddr.ab

Operation: \([\text{dst string}] \leftarrow \text{conversion of } [\text{src string}];\]

Condition \(N \leftarrow [\text{src string}] \text{ LSS 0};\)

Codes: \(Z \leftarrow [\text{src string}] \text{ EQL 0};\)

\(V \leftarrow \{\text{decimal overflow}\};\)

\(C \leftarrow 0;\)

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:

\(R0 = 0\)

\(R1 = \text{address of the byte containing the most significant digit of the source string}\)

\(R2 = 0\)

\(R3 = \text{address of the sign byte of the destination string}\)

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

Purpose: convert leading separate numeric string to packed decimal string

Format: opcode srclen.rw, srcaddr.ab, dstlen.rw, dstaddr.ab

Operation: \( \text{dst string} \leftarrow \text{conversion of src string}; \)

Condition: \( N \leftarrow \text{dst string} \) LSS 0;

Codes: \( Z \leftarrow \text{dst string} \) EQL 0;

V \leftarrow \text{decimal overflow};

C \leftarrow 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:
   - The length of the source Leading Separate Numeric string is outside the range 0 through 31.
   - The length of the destination Packed Decimal String is outside the range 0 through 31.
   - The source string contains an invalid byte. An invalid byte is any character other than an ASCII "0" through "9" in a digit byte or an ASCII "+", "<space>", or "-" in the sign byte.

2. After execution:
   \[ R0 = 0 \]
   \[ R1 = \text{address of the sign byte of the source string} \]
   \[ R2 = 0 \]
   \[ R3 = \text{address of the byte containing the most significant digit of the destination string.} \]

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.

4. 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.
ASHP

ARITHMETIC SHIFT AND ROUND PACKED

Purpose: scale numeric content of a packed decimal string by a power of 10

Format: opcode cnt.rb, srclen.rw, srcaddr.ab, round.rb, dstlen.rw, dstdaddr.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 decently 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

INTRODUCTION

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/1, 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 instruction 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 non-zero 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

275
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:

<table>
<thead>
<tr>
<th>Condition</th>
<th>Codes:</th>
<th>Exceptions:</th>
<th>Opcodes:</th>
<th>Description:</th>
</tr>
</thead>
<tbody>
<tr>
<td>N ←</td>
<td>src string</td>
<td>LSS 0;</td>
<td>reserved operand</td>
<td>38 EDITPC</td>
</tr>
<tr>
<td>Z ←</td>
<td>src string</td>
<td>EQL 0;</td>
<td>decimal overflow</td>
<td></td>
</tr>
<tr>
<td>V ←</td>
<td>decimal overflow</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C ←</td>
<td>significance</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>!N ←</td>
<td>0 if src is −0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>!non-zero digits lost</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Notes:
1. A reserved operand fault occurs with FPD cleared if srclen GTRU 31.

277
2. The destination string is unpredictable if the source string contains an invalid nibble, if the EOSADJUST_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:
   
   \[
   \begin{align*}
   R0 &= \text{length of source string} \\
   R1 &= \text{address of the byte containing the most significant digit of the source string} \\
   R2 &= 0 \\
   R3 &= \text{address of the byte containing the EOSEND pattern operator} \\
   R4 &= 0 \\
   R5 &= \text{address of one byte beyond the last byte of the destination string}
   \end{align*}
   \]

   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 (EOSBLANK_ZERO of EOSREPLACE_SIGN).

7. The contents of the destination string and the memory preceding it are unpredictable if the length covered by EOSBLANK_ZERO or EOSREPLACE_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 \(R3 = \text{location of pattern operator which requested the extra digit. The condition codes and other registers are as specified in note 11. This abort is not continuable.}\)

9. If fewer input digits are requested by the pattern than are specified, then a reserved operand abort is taken with \(R3 = \text{location of EOSEND pattern operator. The condition codes and other registers are as specified in note 11. This abort is not continuable.}\)

10. On an unimplemented or reserved pattern operator, a reserved operand fault is taken with \(R3 = \text{location of the}\

278
faulting pattern operator. The condition codes and other registers are as specified in note 11. This fault is continu-
able 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:

\[
\begin{align*}
N &= \{\text{src has minus sign}\} \\
Z &= \text{all source digits 0 so far} \\
V &= \text{non-zero digits lost} \\
C &= \text{significance} \\
R0<15:0> &= \text{srclen} \\
R0<31:16> &= -\text{number of zeros to supply} \\
R1 &= \text{current source location} \\
R2<7:0> &= \text{fill character} \\
R2<15:8> &= \text{sign character} \\
R2<31:16> &= \text{unpredictable} \\
R3 &= \text{address of edit pattern operator causing exception} \\
R4 &= \text{unpredictable} \\
R5 &= \text{location of next destination byte}
\end{align*}
\]

### SUMMARY OF EDIT PATTERN OPERATORS

<table>
<thead>
<tr>
<th>Insert:</th>
<th>Name</th>
<th>Operand</th>
<th>Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>EO$INSERT</td>
<td>ch</td>
<td>insert character, fill if insignificant</td>
</tr>
<tr>
<td></td>
<td>EO$STORE_SIGN</td>
<td>—</td>
<td>insert sign</td>
</tr>
<tr>
<td></td>
<td>EO$FILL</td>
<td>r</td>
<td>insert fill</td>
</tr>
<tr>
<td>Move:</td>
<td>EO$MOVE</td>
<td>r</td>
<td>move digits, filling insignificant</td>
</tr>
<tr>
<td></td>
<td>EO$FLOAT</td>
<td>r</td>
<td>move digits, floating sign</td>
</tr>
<tr>
<td></td>
<td>EO$END_FLOAT</td>
<td>—</td>
<td>end floating sign</td>
</tr>
<tr>
<td>Fixup:</td>
<td>EO$BLANK-ZERO</td>
<td>len</td>
<td>fill backward when zero replace with fill if -0</td>
</tr>
<tr>
<td></td>
<td>EO$REPLACE_SIGN</td>
<td>len</td>
<td>replace with fill if -0</td>
</tr>
<tr>
<td>Load:</td>
<td>EO$LOAD_FILL</td>
<td>ch</td>
<td>load fill character</td>
</tr>
<tr>
<td></td>
<td>EO$LOAD_SIGN</td>
<td>ch</td>
<td>load sign character</td>
</tr>
<tr>
<td></td>
<td>EO$LOAD_PLUS</td>
<td>ch</td>
<td>load sign character if positive</td>
</tr>
<tr>
<td></td>
<td>EO$LOAD_MINUS</td>
<td>ch</td>
<td>load sign character if negative</td>
</tr>
</tbody>
</table>
**Edit Instruction**

Control:
- EO$SET_SIGNIF set significance flag
- EO$CLEAR_SIGNIF clear significance flag
- EO$ADJUST_INPUT len adjust source length
- EO$END end edit

where:
- \( ch \) = one character
- \( r \) = repeat counter in the range 1 through 15
- \( len \) = length in the range 1 through 255

**EDIT PATTERN OPERATOR ENCODING**

(hex)

<table>
<thead>
<tr>
<th>Decimal</th>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>EO$END</td>
<td></td>
</tr>
<tr>
<td>01</td>
<td>EO$END_FLOAT</td>
<td></td>
</tr>
<tr>
<td>02</td>
<td>EO$CLEAR_SIGNIF</td>
<td></td>
</tr>
<tr>
<td>03</td>
<td>EO$SET_SIGNIF</td>
<td></td>
</tr>
<tr>
<td>04</td>
<td>EO$STORE_SIGN</td>
<td></td>
</tr>
<tr>
<td>05...1F</td>
<td>Reserved to DIGITAL</td>
<td></td>
</tr>
<tr>
<td>20...3F</td>
<td>Reserved for all time</td>
<td></td>
</tr>
<tr>
<td>40</td>
<td>EO$LOAD_FILL</td>
<td></td>
</tr>
<tr>
<td>41</td>
<td>EO$LOAD_SIGN</td>
<td></td>
</tr>
<tr>
<td>42</td>
<td>EO$LOAD_PLUS character is in next byte</td>
<td></td>
</tr>
<tr>
<td>43</td>
<td>EO$LOAD_MINUS</td>
<td></td>
</tr>
<tr>
<td>44</td>
<td>EO$INSERT</td>
<td></td>
</tr>
<tr>
<td>45</td>
<td>EO$BLANK_ZERO</td>
<td></td>
</tr>
<tr>
<td>46</td>
<td>EO$REPLACE_SIGN unsigned length is in next byte</td>
<td></td>
</tr>
<tr>
<td>47</td>
<td>EO$ADJUST_INPUT</td>
<td></td>
</tr>
<tr>
<td>48...5F</td>
<td>Reserved to DIGITAL</td>
<td></td>
</tr>
<tr>
<td>60...7F</td>
<td>Reserved to CSS, customers</td>
<td></td>
</tr>
<tr>
<td>80, 90, A0</td>
<td>Reserved to DIGITAL</td>
<td></td>
</tr>
<tr>
<td>81..8F</td>
<td>EO$FILL</td>
<td></td>
</tr>
<tr>
<td>91..9F</td>
<td>EO$MOVE repeat count is ( &lt;3:0&gt; )</td>
<td></td>
</tr>
<tr>
<td>A1..AF</td>
<td>EO$FLOAT</td>
<td></td>
</tr>
<tr>
<td>B0..FE</td>
<td>Reserved to DIGITAL</td>
<td></td>
</tr>
<tr>
<td>FF</td>
<td>Reserved for all time</td>
<td></td>
</tr>
</tbody>
</table>

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](code)

```plaintext
test
if R0 LEQ 0
then
    begin
    if R0 EQL 0 then [reserved operand];
    READ ← 0;;
    R0 <31:16> ← R0 <31:16> + 1;
    !see EOSADJUST_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):**

```plaintext
(R5) ← char;
R5 ← R5 + 1;
return;
```

Also the following definitions are used:

- fill = R2 <7:0>
- sign = R2 <15:8>
**EO$INSERT**

**INSERT CHARACTER**

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

**Format:** pattern ch

**Operations:** if PSW<C>EQL 1 then STORE (ch) else STORE (fill);

**Pattern Operators:** 44 EO$INSERT

**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.

**Notes:** This pattern operator is used for blankable inserts (e.g., comma) and fixed inserts (e.g., slash). Fixed inserts require that significance be set (by EO$SET_SIGNIF or EO$END_FLOAT).
STORE SIGN

Purpose: Insert the sign character
Format: pattern
Operations: STORE (sign);

Pattern Operators: 04 EO$STORE_SIGN Store Sign
Description: The content of the sign register is placed into the destination.
Notes: This pattern operator is used for any non-floating arithmetic sign. It should be preceded by a EO$LOAD_PLUS and/or EO$LOAD_MINUS if the default sign convention is not desired.
EO$FILL

Purpose: Insert the fill character
Format: pattern r
Operations: repeat r do STORE (fill);
Pattern Operators: 8x EO$FILL Store Fill
Description: The right nibble of the pattern operator is the repeat count. The content of the fill register is placed into the destination repeat times.
Notes: This pattern operator is used for fill (blank) insertion.
MOVE DIGITS

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

Format: pattern  r

Operations: repeat r do
begin
    tmp ← READ;
    if tmp NEQU 0 then 
       begin
            PSW<Z> ← 0;
            PSW<C> ← 1;  !set significance
       end;
    if PSW<C> EQL 0 then STORE (fill)
    else STORE ("0" +tmp);
end;

Pattern Operators: 9x  EO$MOVE  Move Digits

Description: The right nibble of the pattern operator is the repeat count. For
repeat times, the following algorithm is executed. The next
digit is moved from the source to the destination. 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, sig-
nificance must be clear. If leading zero 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.
EO$FLOAT

FLOAT SIGN

Purpose: Move digits, floating the sign across insignificant digits

Format: pattern r

Operations: repeat r do
    begin
        tmp ← READ;
        if tmp NEQU 0 then
            begin
                if PSW<C> EQL 0 then STORE (sign);
                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 signif-
cance is not yet set, then the contents of the sign register are
stored in the destination, significance is set, and zero is
cleared. If the digit is significant, it is stored in the destination,
otherwise the content of the fill register is stored in the destina-
tion.

Notes:

1. If r is greater than the number of digits remaining in the
source string, a reserved operand abort is taken.

2. This pattern operator is used to move digits with a floating
arithmetic sign. The sign must already be set up as for
EO$STORE_SIGN. A sequence of one or more
EO$FLOATs can include intermixed EO$INSERTs and
EO$FILLs. Significance must be clear before the first pat-
tern operator of the sequence. The sequence must be
terminated by one EO$END_FLOAT.

3. This pattern operator is used to move digits with a floating
currency sign. The sign must already be set up with an
EO$LOAD_SIGN. A sequence of one or more
EO$FLOATs can include intermixed EO$INSERTs and
EO$FILLs. Significance must be clear before the first pat-
tern 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

Operations: if PSW<\text{C}> EQL 0 then
begin
  STORE (sign);
  PSW<\text{C}> \leftarrow 1; \quad \text{!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 content of the sign register is stored in the destination and significance is set.

Notes: This pattern operator is used after a sequence of one or more EO\$FLOAT pattern operators which start with significance clear. The EO\$FLOAT sequence can include intermixed EO\$INSERTs and EO\$FILLs.
EO$BLANK_ZERO

BLANK BACKWARDS WHEN ZERO

Purpose: Fix up the destination to be blank when the value is zero

Format: pattern  len

Operations:
if len EQLU 0 then [UNPREDICTABLE];
if PSW<Z> EQL 1 then
begin
  R5 ← R5 − len;
  repeat len do STORE (fill);
end;

Pattern Operators:
45 EO$BLANK_ZERO  Blank Backwards When Zero

Description: The pattern operator is followed by an unsigned byte integer length. If the value of the source string is zero, then the contents of the fill register 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: Fix up the destination sign when the value is minus zero

Format: pattern len

Operations:
if len EQLU 0 then [UNPREDICTABLE];
if PSW<Z> EQL 1 and PSW<N> EQL 1 then
   (R5 - len) ← fill;

Pattern Operators: 46 EO$REPLACE_SIGN Replace Sign When Minus Zero

Description: The pattern operator is followed by an unsigned byte integer length. If the value of the source string is minus zero (i.e., if both N and Z are set), then the content of the 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 unpredictable.
2. This pattern operator is used to correct a stored sign (EO$END_FLOAT or EO$STORE_SIGN) if a minus was stored and the source value turned out to be zero.
**EO$LOAD**

**LOAD REGISTER**

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

**Format:** pattern ch

**Operations:**
- \( \text{fill} \leftarrow \text{ch}; \) \hspace{1cm} !EO$LOAD\_FILL
- \( \text{sign} \leftarrow \text{ch}; \) \hspace{1cm} !EO$LOAD\_SIGN
- if PSW\( <N> \) EQL 0 then sign \( \leftarrow \) ch; \hspace{1cm} !EO$LOAD\_PLUS
- if PSW\( <N> \) EQL 1 then sign \( \leftarrow \) ch; \hspace{1cm} !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\_MINUS this character is placed into the sign register if the source string has a negative sign.

**Notes:**
1. EO$LOAD\_FILL is used to set up check protection (* instead of space).
2. EO$LOAD\_SIGN is used to set up a floating currency sign.
3. EO$LOAD\_PLUS is used to set up a non-blank plus sign.
4. EO$LOAD\_MINUS is used to set up a non-minus minus sign (such as CR, DB, or the PL/1+).
SIGNIFICANCE

Purpose: Control the significance (leading zero) indicator

Format: pattern

Operations: 
- PSW< C > ← 0; !EO$CLEAR_SIGNIF
- PSW< C > ← 1; !EO$SET_SIGNIF

Pattern Operators: 
- 02 EO$CLEAR_SIGNIF Clear Significance
- 03 EO$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$SET_SIGNIF is used to avoid leading suppression (before EO$MOVE) or to force a fixed insert (before EO$INSERT).
**EO$ADJUST_INPUT**

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

**Format:** pattern  len

**Operations:**

if len EQLU 0 or len GTRU 31 then [UNPREDICTABLE];
if R0 <15:0> GTRU len
then
begin
  R0<31:16> ← 0
  repeat R0 <15:0> ← len do
    if READ NEQU 0 then
      begin
        PSW<Z> ← 0;
        PSW<V> ← 1;
      end;
    end;
  else R0<31:16> ← R0<15:0> ← len;
end;

**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 non-zero then overflow 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>.

**Notes:** 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
Operations: exit_flag ← true;
            !terminate edit loop
            !end processing is
            !describe under EDITPC instruction

Pattern Operators:

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

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 cur-
tently 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 sys-
tem 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 the respective
VAX-11 Hardware Handbook.

Exceptions are handled by the operating system. Usually, they are
reflected to the originating mode as a signal. (Refer to Appendix C.) In
general, the exception is described by 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 instruc-
tion 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 con-
ditions with a single instruction; see the BISPSW and BICPSW
instruction described in Chapter 7.
Exceptions

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

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

PROCESSOR STATUS

When an exception or an interrupt is serviced, the processor status must be preserved so that the interrupted process may continue normally. 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 the Privilege Instruction chapter of the appropriate VAX-11 Hardware Handbook.) Other processor status is changed even less frequently. (Again, refer to the processor internal register descriptions in the Privilege Instruction chapter of the appropriate VAX-11 Hardware Handbook.)

The Processor Status Longword (PSL) is a longword consisting of a word of privileged processor status concatenated with the Processor Status Word (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 \textless 31:21\textgreater 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. The Processor Status Longword is illustrated in Figure 12-1.
Exceptions

![PSW](image)

**Figure 12-1  Processor Status Longword**

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

**BITS**

**DESCRIPTION**

3:0

Condition Codes: N, Z, V, C

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 (PRV_MOD). Loaded from current access mode by exceptions and CHMX instructions, cleared by interrupts, and restored by REI.

25:24  Current Access Mode (CUR_MOD). 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.
Exceptions

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

The software mnemonics for the PSL fields are:

<table>
<thead>
<tr>
<th>MNEMONIC</th>
<th>VALUE</th>
<th>MEANING</th>
</tr>
</thead>
<tbody>
<tr>
<td>PSL$V_TBIT</td>
<td>4</td>
<td>position of trace enable</td>
</tr>
<tr>
<td>PSL$M_TBIT</td>
<td>1 @ 4</td>
<td>mask for trace enable</td>
</tr>
<tr>
<td>PSL$V_IV</td>
<td>5</td>
<td>position of IV enable</td>
</tr>
<tr>
<td>PSL$M_IV</td>
<td>1 @ 5</td>
<td>mask for IV enable</td>
</tr>
<tr>
<td>PSL$V_FU</td>
<td>6</td>
<td>position of FU enable</td>
</tr>
<tr>
<td>PSL$M_FU</td>
<td>1 @ 6</td>
<td>mask for FU enable</td>
</tr>
<tr>
<td>PSL$V_DV</td>
<td>7</td>
<td>position of DV enable</td>
</tr>
<tr>
<td>PSL$M_DV</td>
<td>1 @ 7</td>
<td>mask for DV enable</td>
</tr>
<tr>
<td>PSL$V_IPL</td>
<td>16</td>
<td>position of IPL</td>
</tr>
<tr>
<td>PSL$S_IPL</td>
<td>5</td>
<td>size of IPL</td>
</tr>
<tr>
<td>PSL$V_PRVMOD</td>
<td>22</td>
<td>position of PRVMOD</td>
</tr>
<tr>
<td>PSL$S_PRVMOD</td>
<td>2</td>
<td>size of PRVMOD</td>
</tr>
<tr>
<td>PSL$V_CURMOD</td>
<td>24</td>
<td>position of CURMOD</td>
</tr>
<tr>
<td>PSL$S_CURMOD</td>
<td>2</td>
<td>size of CURMOD</td>
</tr>
<tr>
<td>PSL$V_IS</td>
<td>26</td>
<td>position of IS bit</td>
</tr>
<tr>
<td>PSL$M_IS</td>
<td>1 @ 26</td>
<td>mask for IS bit</td>
</tr>
<tr>
<td>PSL$V_FPD</td>
<td>27</td>
<td>position of FPD bit</td>
</tr>
<tr>
<td>PSL$M_FPD</td>
<td>1 @ 27</td>
<td>mask for FPD bit</td>
</tr>
<tr>
<td>PSL$V_TP</td>
<td>30</td>
<td>position of TP bit</td>
</tr>
<tr>
<td>PSL$M_TP</td>
<td>1 @ 30</td>
<td>mask for TP bit</td>
</tr>
<tr>
<td>PSL$V_CM</td>
<td>31</td>
<td>position of CM bit</td>
</tr>
<tr>
<td>PSL$M_CM</td>
<td>1 @ 31</td>
<td>mask for CM bit</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>MNEMONIC</th>
<th>VALUE</th>
<th>MEANING</th>
</tr>
</thead>
<tbody>
<tr>
<td>PSL$K_KERNEL</td>
<td>0</td>
<td>kernel mode</td>
</tr>
<tr>
<td>PSL$K_EXEC</td>
<td>1</td>
<td>executive mode</td>
</tr>
<tr>
<td>PSL$K_SUPER</td>
<td>2</td>
<td>supervisor mode</td>
</tr>
<tr>
<td>PSL$K_USER</td>
<td>3</td>
<td>user mode</td>
</tr>
</tbody>
</table>

**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 condition identification when signaled. Each of them indicates that an exception had occurred.
Exceptions

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

<table>
<thead>
<tr>
<th>TYPE CODE</th>
<th>TRAP TYPE</th>
<th>SOFTWARE MNEMONIC</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>integer overflow</td>
<td>SRM$K_INT_OVF_T</td>
</tr>
<tr>
<td>2</td>
<td>integer divide by zero</td>
<td>SRM$K_INT_DIV_T</td>
</tr>
<tr>
<td>3</td>
<td>floating overflow</td>
<td>SRM$K_FLT_OVF_T</td>
</tr>
<tr>
<td>4</td>
<td>floating/decimal divide by zero</td>
<td>SRM$K_FLT_DIV_T</td>
</tr>
<tr>
<td>5</td>
<td>floating underflow</td>
<td>SRM$K_FLT_UND_T</td>
</tr>
<tr>
<td>6</td>
<td>decimal overflow</td>
<td>SRM$K_DEC_OVF_T</td>
</tr>
<tr>
<td>7</td>
<td>subscript range</td>
<td>SRM$K_SUB_RNG_T</td>
</tr>
</tbody>
</table>

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.

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).

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
Exceptions

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$KFLT_OVF_T).

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$KFLT_DIV_T).

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 underflow. The condition codes are set on the final result in POLYx. The type code pushed on the stack is 5 (SRM$KFLT_UND_T).

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).

Subscript Range Trap
A subscript range trap is an exception that indicates that the last instruction was an INDEK 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).
Exceptions

Floating Overflow Fault
A floating overflow fault is an exception that indicates that the last instruction executed resulted in an exponent greater than the largest representable exponent for the data type after normalization and rounding. The destination was unaffected and the saved condition codes are UNPREDICTABLE. The saved PC points to the instruction causing the fault. In the case of a POLY instruction, the instruction is suspended with FPD set. The type code pushed on the stack is 8 (SRM$K_FLT_OVF_F).

Divide by Zero Floating Fault
A floating divide by zero fault is an exception that indicates that the last instruction executed had a floating zero divisor. The quotient operand was unaffected and the saved condition codes are UNPREDICTABLE. The saved PC points to the instruction causing the fault. In the case of a POLY instruction, the instruction is suspended with FPD set. The type code pushed on the stack is 9 (SRM$K_FLT_DIV_F).

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

EXCEPTIONS DETECTED DURING OPERAND REFERENCE

Access Control Violation Fault
An access control violation fault is an exception indicating that the process attempted a reference not allowed in the access mode at which the process was operating. (Refer to the appropriate VAX-11 Hardware Handbook for a description of the information pushed on the stack as parameters.) Software may restart the process after changing the address translation information.

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. (Refer to the appropriate VAX-11 Hardware Handbook 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.
Exceptions

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:

<table>
<thead>
<tr>
<th>SPECIFIER</th>
<th>RESERVED SITUATION</th>
</tr>
</thead>
<tbody>
<tr>
<td>Short Literal</td>
<td>Modify, destination, address source, or within index mode.</td>
</tr>
<tr>
<td>Register</td>
<td>Address source or within index mode.</td>
</tr>
<tr>
<td>Index Mode</td>
<td>Within index mode, or with PC as index.</td>
</tr>
</tbody>
</table>

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), and (PC)+ as a write destination.

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)

**EXCEPTIONS OCCURRING AS THE CONSEQUENCE OF AN INSTRUCTION START**

**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.

**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. No attempt is made by DIGITAL to register customer usage.

**Compatibility Mode Exception**
A compatibility mode exception is an exception that occurs when the processor is in compatibility mode. (Refer to the appropriate VAX-11 Hardware Handbook.)

**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 exception will occur. 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.

TRACING
A trace is an exception that occurs between instructions when Trace is enabled. Tracing is used for tracing programs, for performance evaluation, or for 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). If only one bit were used, the occurrence of an interrupt at the end of the instruction would either produce no traces 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.

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.
Exceptions

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.

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.

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:

<table>
<thead>
<tr>
<th></th>
<th>enabled at beg (T)</th>
<th>enabled at end (T)</th>
<th>TP bit at end (TP)</th>
</tr>
</thead>
<tbody>
<tr>
<td>XXX</td>
<td>N</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td></td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>CHMx...REI</td>
<td>N</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td></td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>interrupt or exception...REI</td>
<td>N</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td></td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>CALLx</td>
<td>N</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td></td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
</tbody>
</table>

(pushed PSW<T> cleared)
### Exceptions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>N</th>
<th>N*</th>
<th>N</th>
</tr>
</thead>
<tbody>
<tr>
<td>RET</td>
<td>N</td>
<td>Y*</td>
<td>N (trap after next instruction)</td>
</tr>
<tr>
<td></td>
<td>Y</td>
<td>N*</td>
<td>Y</td>
</tr>
<tr>
<td></td>
<td>Y</td>
<td>Y*</td>
<td>Y</td>
</tr>
<tr>
<td>CHMx</td>
<td>N</td>
<td>N</td>
<td>N (pushed PSL&lt;TP&gt; clear)</td>
</tr>
<tr>
<td></td>
<td>Y</td>
<td>N</td>
<td>N (pushed PSL&lt;TP&gt; set)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Instruction</th>
<th>N</th>
<th>N*</th>
<th>N</th>
</tr>
</thead>
<tbody>
<tr>
<td>REI</td>
<td>N</td>
<td>Y*</td>
<td>N</td>
</tr>
<tr>
<td>REI (if PSL&lt;TP&gt;=0 on stack)</td>
<td>N</td>
<td>Y*</td>
<td>N</td>
</tr>
<tr>
<td></td>
<td>Y</td>
<td>N*</td>
<td>Y</td>
</tr>
<tr>
<td></td>
<td>Y</td>
<td>Y*</td>
<td>Y</td>
</tr>
<tr>
<td>REI (if PSL&lt;TP&gt;=1 on stack)</td>
<td>N</td>
<td>Y*</td>
<td>Y</td>
</tr>
<tr>
<td></td>
<td>Y</td>
<td>N*</td>
<td>Y</td>
</tr>
<tr>
<td></td>
<td>Y</td>
<td>Y*</td>
<td>Y (only one trap)</td>
</tr>
<tr>
<td>BISP SW</td>
<td>N</td>
<td>Y</td>
<td>N</td>
</tr>
<tr>
<td></td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>BICPSW</td>
<td>N</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td></td>
<td>Y</td>
<td>N</td>
<td>Y</td>
</tr>
<tr>
<td>interrupt or exception</td>
<td>N</td>
<td>N</td>
<td>N (pushed PSL&lt;TP&gt; clear)</td>
</tr>
<tr>
<td></td>
<td>Y</td>
<td>N</td>
<td>N (pushed PSL&lt;TP&gt; depends on above description)</td>
</tr>
</tbody>
</table>

* = depends on PSW<T> popped from stack

### 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.

307
Exceptions

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.

SERIOUS SYSTEM FAILURES
Refer to the appropriate VAX-11 Hardware Handbook for details on the following failures.

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.

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.

Machine Check Exception
A machine check exception indicates that the processor detected an internal error in itself.
STACKS
At any time, the processor is either in a process context (IS = 0) in one of four modes (kernel, executive, supervisor, user), or in the system-wide interrupt service context (IS = 1) that operates with kernel privileges. There is a stack pointer associated with each of these five states, and any time the processor changes from one of these states to another, SP (R14) is stored in the process context stack pointer for the old state and loaded from that for the new state. The process context stack pointers (KSP = kernel, ESP = executive, SSP = supervisor, USP = user) are allocated in the Process Control Block, although some hardware implementations may keep them in internal registers. The interrupt stack pointer (ISP) is in an internal register.

Stack Residency
The user, supervisor, and executive 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.

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.
Exceptions

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:

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

The processor does not allow 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 a 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.
RELATED INSTRUCTIONS

RETURN FROM EXCEPTION OR INTERRUPT

Purpose: exit from an exception or interrupt service routine
Format: Opcode
Operation:

```
tmp1 ← (SP) +;  !Pick up saved PC
tmp2 ← (SP) +;  !and PSL
if \[(tmp2<current_mode> LSS\{current_mode\}) or
    \{tmp2<IS> EQLU 1 and PSL<IS>EQLU 0\} or
    \{tmp2<IS> EQLU 1 and
    tmp2<current_mode> NEQU 0\} or
    \{tmp2<IS> EQLU 1 and tmp2<IPL> EQLU 0\} or
    \{tmp2<IPL> GRTU 0 and
    tmp2<current_mode> NEQU 0\} or
    \{tmp2<previous_mode> LSSU
    tmp2<current_mode>\} or
    \{tmp2<IPL> GTRU PSL<IPL>\} or
    \{tmp2<PSL_M8Z> 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
        Icheck for AST delivery
    then [request interrupt at IPL 2];
end;
[clear instruction buffer]:
```

Condition Codes:

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

311
Exceptions

Exceptions: reserved operand

Opcodes: 02 REI Return from Exception or Interrupt

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

Notes:

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

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

3. The 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. 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: N ← 0;
Codes:    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 N ← 0; !If reserved to DIGITAL opcode fault
Codes: 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.
Notes: This opcode is 0 to trap many branches to data.
CHAPTER 13
PDP-11 COMPATIBILITY MODE

INTRODUCTION
Compatibility Mode is provided in the VAX architecture to allow a certain subset of PDP-11 programs to be directly executed on VAX machines. VAX compatibility mode hardware, in conjunction with a compatibility mode software executive (which runs in VAX mode), can emulate the environment provided to user programs on a PDP-11. This environment excludes from a complete PDP-11 the normal operation of the following features:

1. Privileged instructions such as HALT and RESET.
2. Special instructions such as traps and WAIT.
3. Access to internal processor registers (e.g., PSW and console switch register).
4. Direct access to trap and interrupt vectors.
5. Direct access to I/O devices. (Compatibility mode programs can directly reference I/O devices if and only if proper mapping has been established by VAX mode software.)
6. Interrupt servicing.
7. Stack overflow protection.
8. Alternate general register sets.
9. An PDP-11 processor modes other than user (i.e., kernel and supervisor modes are not supported).
10. Floating point instructions.

The VAX-11 compatibility mode architecture is split into two parts. The first part, described briefly in this chapter, is the PDP-11 environment provided by the VAX compatibility mode hardware. Details of the operation of PDP-11 compatible operations can be found in the appropriate PDP-11 handbook. The second part is the hardware mechanisms provided in the VAX architecture that enable the implementation of various compatibility mode executives, which is part of the VAX-11 System Architecture.

COMPATIBILITY MODE USER ENVIRONMENT

General Registers And Address Modes
All of the PDP-11 general registers and addressing modes are provided in compatibility mode. Side effects caused by a destination address
calculation have no effect on source values, and auto-increment modes in JMP and JSR do not affect the new PC. All addresses are 16 bits wide.

The Stack
General register R6 is used as the stack pointer by certain instructions, as in the PDP-11. It is not, however, used by the hardware for any exceptions or interrupts. There is also no stack overflow protection in compatibility mode.

Processor Status Word
A subset of the full PDP-11 Processor Status Word is available in compatibility mode. The format of the compatibility mode PSW is:

<table>
<thead>
<tr>
<th>15</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>T</td>
<td>N</td>
<td>Z</td>
<td>V</td>
<td>C</td>
<td></td>
</tr>
</tbody>
</table>

The PSW can only be affected by the condition code instructions, RTI, and RTT. When an RTI or RTT instruction is executed, bits 15 through 5 in the saved PSW on the stack are ignored.

Instructions
The following instructions are provided by the compatibility mode hardware.

Table 1 Compatibility Mode Instructions

<table>
<thead>
<tr>
<th>Opcode (octal)</th>
<th>Mnemonic</th>
</tr>
</thead>
<tbody>
<tr>
<td>000002</td>
<td>RTI</td>
</tr>
<tr>
<td>000006</td>
<td>RTT</td>
</tr>
<tr>
<td>0001DD</td>
<td>JMP</td>
</tr>
<tr>
<td>00020R</td>
<td>RTS</td>
</tr>
<tr>
<td>000240-000277</td>
<td>Condition Codes</td>
</tr>
<tr>
<td>0003DD</td>
<td>SWAB</td>
</tr>
<tr>
<td>000400-003777</td>
<td>Branches</td>
</tr>
<tr>
<td>100000-103777</td>
<td>Branches</td>
</tr>
<tr>
<td>004RDD</td>
<td>JSR</td>
</tr>
<tr>
<td>.050DD</td>
<td>CLR(B)</td>
</tr>
<tr>
<td>.051DD</td>
<td>COM(B)</td>
</tr>
<tr>
<td>.052DD</td>
<td>INC(B)</td>
</tr>
<tr>
<td>.053DD</td>
<td>DEC(B)</td>
</tr>
</tbody>
</table>
PDP-11 Compatibility Mode

.054DD       NEG(B)
.055DD       ADC(B)
.056DD       SBC(B)
.057DD       TST(B)
.060DD       ROR(B)
.061DD       ROL(B)
.062DD       ASR(B)
.063DD       ASL(B)
.0065SS      MFPI*
.0066DD      MTPI*
.1065SS      MFPD*
.1066DD      MTPD*
.0067DD      SXT
.070RSS      MUL
.071RSS      DIV
.072RSS      ASH
.073RSS      ASHC
.074RSS      XOR
.077RNN      SOB
.1SSDD       MOV(B)
.2SSDD       CMP(B)
.3SSDD       BIT(B)
.4SSDD       BIC(B)
.5SSDD       BIS(B)
.06SSDD      ADD
.16SSDD      SUB

* These instructions execute exactly as they would on a PDP-11 in user mode with Instruction and Data space overmapped. More specifically, they ignore the previous access level and act like PUSH and POP instructions referencing the current stack.

The following trap instructions cause the machine to enter VAX mode, where either the complete trap may be serviced, or where just the instruction may be simulated.
Table 2  Compatibility Mode Trap Instructions

<table>
<thead>
<tr>
<th>Opcode (octal)</th>
<th>Mnemonic</th>
</tr>
</thead>
<tbody>
<tr>
<td>000003</td>
<td>BPT</td>
</tr>
<tr>
<td>000004</td>
<td>IOT</td>
</tr>
<tr>
<td>104000-104377</td>
<td>EMT</td>
</tr>
<tr>
<td>104400-104777</td>
<td>TRAP</td>
</tr>
</tbody>
</table>

The following instructions and all other opcodes not defined above are considered reserved instructions in compatibility mode, and trap to VAX mode.

Table 3  Compatibility Mode Reserved Instructions

<table>
<thead>
<tr>
<th>Opcode (octal)</th>
<th>Mnemonic</th>
</tr>
</thead>
<tbody>
<tr>
<td>000000</td>
<td>HALT</td>
</tr>
<tr>
<td>000001</td>
<td>WAIT</td>
</tr>
<tr>
<td>000005</td>
<td>RESET</td>
</tr>
<tr>
<td>00023N</td>
<td>SPL</td>
</tr>
<tr>
<td>0064NN</td>
<td>MARK</td>
</tr>
<tr>
<td>07500R</td>
<td>FADD—FIS</td>
</tr>
<tr>
<td>07501R</td>
<td>FSUB—FIS</td>
</tr>
<tr>
<td>07502R</td>
<td>FMUL—FIS</td>
</tr>
<tr>
<td>07503R</td>
<td>FDIV—FIS</td>
</tr>
<tr>
<td>17XXXX</td>
<td>FP11 Floating Point</td>
</tr>
</tbody>
</table>

Note that no floating point instructions are included in compatibility mode.

ENTERING AND LEAVING COMPATIBILITY MODE
Compatibility mode is entered by executing an REI instruction with the compatibility mode bit set in the image of the PSL on the stack. Other bits in the PSL have the following effects:

<table>
<thead>
<tr>
<th>NZVC</th>
<th>T Bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>DV</td>
<td>Reserved operand fault if not zero</td>
</tr>
<tr>
<td>FU</td>
<td>Reserved operand fault if not zero</td>
</tr>
<tr>
<td>IV</td>
<td>Reserved operand fault if not zero</td>
</tr>
<tr>
<td>IPL</td>
<td>Reserved operand fault if not zero</td>
</tr>
<tr>
<td>PRV MOD</td>
<td>Reserved operand fault if not 3</td>
</tr>
<tr>
<td>CUR MOD</td>
<td>Reserved operand fault if not 3</td>
</tr>
<tr>
<td>IS</td>
<td>Reserved operand fault if not zero</td>
</tr>
</tbody>
</table>
PDP-11 Compatibility Mode

FPD  Reserved operand fault if not zero
TP   T pending bit.

VAX mode is re-entered from compatibility mode by the compatibility mode program causing an exception, or by an interrupt. The PSL pushed on the kernel or interrupt stack when leaving compatibility mode has all the bits that cause reserved operand faults in the above table set to the appropriate state.

Note that when an RTI or RTT instruction is executed in compatibility mode, the 11 high bits of the PSW are ignored, but when the PSW is restored as part of the PSL when going from VAX to compatibility mode, those bits must be zero, or a reserved operand fault occurs.

General Register Usage
Compatibility mode registers R0 through R6 are bits 15 through 0, of VAX general registers R0 through R6, respectively. Compatibility mode register R7 (PC) is bits 15 through 0 of VAX general register R15 (PC). VAX registers R8 through R14 (SP) are not affected by compatibility mode. When entering compatibility mode, VAX register R7 and the upper halves of registers R0 through R6 and R15 are ignored. When an exception or interrupt occurs from compatibility mode, VAX register R7 is UNPREDICTABLE and the upper halves of R0 through R6 and the stacked R15 (PC) are zero. Since there are no FP11 floating point instructions in compatibility mode, there are no floating accumulators.

COMPATIBILITY MODE MEMORY MANAGEMENT
PDP-11 addresses are 16-bit byte addresses, hence compatibility mode programs are confined to execute in the first 64 Kb of the per process part of the virtual address space. There is a one-to-one correspondence between a compatibility mode virtual address and its VAX counterpart (e.g., virtual address 0 references the same location in both modes). A compatibility mode address is interpreted as follows:

<table>
<thead>
<tr>
<th>31</th>
<th>16</th>
<th>15</th>
<th>9</th>
<th>8</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>PAGE</td>
<td>DISPLACEMENT</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The PDP-11 capability of providing different access protection to different segments is automatically provided since protection is specified at the page level in the VAX architecture (i.e., VAX pages are smaller than PDP-11 segments).
The memory management system protects and relocates compatibility mode addresses in the normal manner. Thus, all of the memory management mechanisms available in VAX mode are available to the compatibility mode executive for managing both the virtual and physical memory of compatibility mode programs. All of the exception conditions which can be caused by memory management in VAX mode can also occur when relocating a compatibility mode address.

Most of the features of the KT11-D affecting the user environment can be simulated with the VAX memory management system. The following table provides a general description of how this can be done; reference the appropriate VAX hardware handbook and the appropriate PDP-11 handbooks for details of each system.

<table>
<thead>
<tr>
<th>KT11-D Feature to be Simulated</th>
<th>VAX Simulation Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>Eight segments per user.</td>
<td>Eight segments can be simulated by dividing the 128 pages of the compatibility mode virtual address space into eight logical groups of 16 pages each having possibly different protection.</td>
</tr>
<tr>
<td>Segment size from 64 to 8K bytes (1 to 128 blocks) in 64 byte increments, using contiguous memory.</td>
<td>Segment size from 512 to 8K bytes (1 to 16 pages) in 512 byte (1 page) increments, using discontiguous memory.</td>
</tr>
<tr>
<td>Forward growing segments (Expand Direction=0).</td>
<td>Can be simulated using page table entries specifying no access for those pages that are not allocated.</td>
</tr>
<tr>
<td>Backward growing segments (ED=1).</td>
<td>Can be simulated using page table entries specifying no access for those pages that are not allocated.</td>
</tr>
<tr>
<td>Segments begin on any 64 byte boundary.</td>
<td>Segments begin on any 512 byte boundary.</td>
</tr>
</tbody>
</table>

What follows is an example of how a PDP-11 environment can be created using the above concepts. Segments 0, 1, and 2 or the PDP-11 environment are program segments; 3 is unused; 4 and 5 are stack; and 6 and 7 are read/write data.
**PDP-11 Compatibility Mode**

<table>
<thead>
<tr>
<th>Eleven Environment</th>
<th>VAX Page Table</th>
</tr>
</thead>
<tbody>
<tr>
<td>Size (bytes)</td>
<td>Expand</td>
</tr>
<tr>
<td>0</td>
<td>8K</td>
</tr>
<tr>
<td>1</td>
<td>8K</td>
</tr>
<tr>
<td>2</td>
<td>256</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>1K</td>
</tr>
<tr>
<td>5</td>
<td>8K</td>
</tr>
<tr>
<td>6</td>
<td>8K</td>
</tr>
<tr>
<td>7</td>
<td>2K</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**COMPATIBILITY MODE EXCEPTIONS AND INTERRUPTS**
All interrupts and exception conditions which occur while the machine is in compatibility mode cause the machine to enter VAX mode, and are serviced as indicated in Chapter 12 (note that this includes backing up instruction side effects if necessary). The following exception conditions are specific to compatibility mode. All these exceptions create a three longword frame on the kernel stack containing PSL, PC, and one longword of trap-specific information. Bits 15 through 0 of this longword contain a code indicating the specific type of trap and bits 31 through 16 are zero.

**Reserved Instruction Trap**
These are the opcodes that are defined in compatibility mode. The code for the reserved instruction trap is 0.

**BPT Instruction**
The code for the BPT instruction is 1.

**IOT Instruction**
The code for the IOT instruction is 2.

**EMT Instruction**
The code for the group of EMT instructions is 3.

**TRAP Instruction**
The code for the group of TRAP instructions is 4.

**Illegal Instructions**
Illegal instructions in compatibility mode are JMP and JSR instructions with a register destination. The code for illegal instructions is 5.
**Odd Address Error**

An odd address error trap is caused in compatibility mode whenever a word reference is attempted on a byte boundary. References that use the SP or PC are always word references, even if used in a byte instruction. The code for odd address errors is 6.

**T Bit Operation in Compatibility Mode**

A compatibility mode T bit trap occurs at the end of an instruction when the T bit is set in the PSW at the beginning of the instruction. A T bit trap also occurs at the end of an RTI instruction if the T bit was set in the PSW POPed from the stack. On T bit traps, a 2-longword kernel stack frame is created, containing the PSL and PC. IPL and IS are zero and CM is one in the stacked PSL. Compatibility mode T bit trap uses the same vector as VAX mode T bit trap. (Refer to Chapter 12, Exceptions.)

There are two ways to get the T bit set at the beginning of a compatibility mode instruction.

1. An RTT instruction is executed in compatibility mode and the T bit is set in the PSW image on the stack. In this case, the next instruction is executed (the one pointed to by the PC on the stack), and a T bit trap is taken after that instruction.

2. An REI instruction is executed in VAX mode which has both the T bit and CM bit set (and T pending clear) in the saved PSL image on the stack. Again, one instruction is executed, and the T bit trap is taken. (For a complete description of the interaction of REI, T-bit, and T pending, see Chapter 12. The operations that occur as a function of these conditions are the same whether or not compatibility mode is being entered from the REI.)

The T bit interacts with other compatibility mode operations as follows:

1. T bit set at the beginning of a compatibility mode instruction which does not trap.

In this case, a T bit trap is taken after the instruction. The saved PSL has the T bit set and TP clear. The compatibility mode executive will do one of the following things:

- If it services the trap directly, it may clear the T bit in the saved PSL on the kernel stack if it no longer wants to trace the program, or it may leave it set if it wants to continue tracing the program. It exits with an REI.

- If it returns the trap to compatibility mode, it pushes a (16-bit) PC and (16-bit) PSW with the T bit set on the User stack to simulate the effect of the T bit trap. It then clears the T bit in
the saved PSL image on the kernel stack, and does an REI. The compatibility mode service routine then may clear the T bit in the PSW image on its stack, as a function of whether or not it wants to continue tracing. The compatibility mode routine returns with RTT. (If it always clears the T bit in the saved PSW, it does not matter if it returns with RTI or RTT.)

2. T bit set at the beginning of an RTI or RTT.

A T bit trap occurs immediately after the instruction is executed. There are two different cases, depending on whether or not the T bit was set in the image of the PSW which was popped from the stack by the instruction:

- T bit not set.
  
  Neither TP nor T will be set in the saved PSL on the kernel stack.

- T bit set.

  TP will not be set, and T will be set. This is the same case as for nontrapping instructions.

3. T bit set at the beginning of any instruction which causes a compatibility mode trap.

The trap condition is serviced first. TP is set in the saved PSL pushed on the kernel stack. These traps may be serviced in one of two ways:

- The compatibility mode executive directly services the trap condition.

  In this case, when the compatibility mode executive is done, it executes an REI. The TP and CM bits in the PSL image on the stack will be set, so a compatibility mode T bit trap will immediately be taken.

- The compatibility mode executive returns the trap condition to a compatibility mode routine which services the trap.

In this case, the compatibility mode executive will push a (16-bit) PC and (16-bit) PSW on the user stack to simulate the effect of the trap. The PSW pushed by the compatibility mode executive will have the T bit set, because the TP bit was set in the saved PSL on the kernel stack. The compatibility mode executive will then clear the T and TP bits in the saved PSL and do an REI to the compatibility mode service routine. When the compatibility mode routine is done servicing the trap, it will do an RTI, which will then cause the compatibility mode T bit trap to occur.

325
UNIMPLEMENTED PDP-11 TRAPS
There are several traps that occur in PDP-11s that are not implemented in compatibility mode:

1. There is no stack overflow trap. This is equivalent to the User Mode of the KT11, where there is also no overflow protection. Stack overflow can be provided by the compatibility mode executive using the memory management mechanisms.

2. There is no concept of a double error trap in compatibility mode, since the first error always puts the machine in VAX mode.

3. All other trap conditions such as power failure, memory parity, and memory management traps cause the machine to enter VAX mode.

COMPATIBILITY MODE I/O REFERENCES
Since I/O devices are accessible with all instructions in VAX mode (as in the PDP-11), I/O devices may be referenced directly from compatibility mode, if the memory mapping is set up to allow it. This may be done by mapping pages directly to I/O devices. Note that, in general, I/O devices will not appear in the physical address space on VAX machines the same way they do on PDP-11s, so existing PDP-11 programs that directly reference I/O devices probably will not work. In addition, compatibility mode programs can only do word or byte references; many VAX I/O devices may require that some references be 32 bits wide.

PROCESSOR REGISTERS
The only processor register available in compatibility mode is part of the PSW, and it may only be referenced with the condition code instructions, RTI, and RTT. Access to all other registers must be done in VAX mode.

PROGRAM SYNCHRONIZATION
All PDP-11s guarantee that read-modify-write operations to I/O device registers are interlocked; that is, the device can determine at the time of the read that the same register will be written as the next bus cycle. This synchronization also works in memory on most PDP-11s. In compatibility mode, instructions that have modify destinations will perform this synchronization for UNIBUS I/O device registers and never for memory.
APPENDIX A
DATA TABLES

INTRODUCTION
This appendix contains the following information:
• Hexadecimal-to-decimal conversion
• Decimal-to-hexadecimal conversion
• Hexadecimal addition
• Hexadecimal multiplication
• ASCII* character set
• Hexadecimal-ASCII conversion
• Powers of 2
• Powers of 16

HEXADECIMAL-TO-DECIMAL CONVERSION
For each integer position of the hexadecimal value, locate the corresponding column integer in Table A-1 and record its decimal equivalent in the conversion table. Add the decimal equivalents to obtain the decimal value.

Example:

\[
\begin{align*}
D0500AD0(16) & = ?(10) \\
D000000 & = 3,489,660,928 \\
500000 & = 5,242,880 \\
A00 & = 2,560 \\
D0 & = 208 \\
\hline
D0500AD0 & = 3,494,904,576
\end{align*}
\]

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 be converted.
2. Record the hexadecimal equivalent followed by the number of zeros (0) that corresponds to the integer column minus one.
Appendix A

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:

\[
\begin{array}{ccc}
22,466(10) & = & ?(16) \\
20,480 & = & 5000 & 22,466 \\
1,792 & = & 700 & -20,480 \\
192 & = & C0 & \\
2 & = & 2 & 1,986 \\
& = & & -1,792 \\
\hline
22,466 & = & 57C2 & 194 \\
& & & -192 \\
\hline
& & & \\
& & & 2 \\
& & & -2 \\
\hline
& & & 0
\end{array}
\]

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).

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 \times A, locate 4 in the lefthand column and A in the topmost row. The intersection of the two is Z8 (hex) which is the product of the two numbers.

328
<table>
<thead>
<tr>
<th>HEX</th>
<th>DEC</th>
<th>HEX</th>
<th>DEC</th>
<th>HEX</th>
<th>DEC</th>
<th>HEX</th>
<th>DEC</th>
<th>HEX</th>
<th>DEC</th>
<th>HEX</th>
<th>DEC</th>
<th>HEX</th>
<th>DEC</th>
<th>HEX</th>
<th>DEC</th>
<th>HEX</th>
<th>DEC</th>
<th>HEX</th>
<th>DEC</th>
<th>HEX</th>
<th>DEC</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>16</td>
<td>777</td>
<td>1</td>
<td>268</td>
<td>435</td>
<td>456</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>268</td>
<td>435</td>
<td>456</td>
<td>1</td>
<td>16</td>
<td>777</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>4096</td>
<td>1</td>
<td>256</td>
<td>1</td>
<td>16</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>536</td>
<td>870</td>
<td>912</td>
<td>2</td>
<td>33</td>
<td>554</td>
<td>432</td>
<td>2</td>
<td>2</td>
<td>0</td>
<td>1</td>
<td>576</td>
<td>1</td>
<td>313</td>
<td>072</td>
<td>2</td>
<td>8</td>
<td>192</td>
<td>2</td>
<td>512</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>805</td>
<td>306</td>
<td>368</td>
<td>3</td>
<td>50</td>
<td>331</td>
<td>648</td>
<td>3</td>
<td>3</td>
<td>1</td>
<td>45</td>
<td>728</td>
<td>3</td>
<td>196</td>
<td>608</td>
<td>3</td>
<td>12</td>
<td>288</td>
<td>3</td>
<td>768</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>1073</td>
<td>371</td>
<td>824</td>
<td>4</td>
<td>67</td>
<td>108</td>
<td>864</td>
<td>4</td>
<td>4</td>
<td>1</td>
<td>94</td>
<td>304</td>
<td>4</td>
<td>262</td>
<td>144</td>
<td>4</td>
<td>16</td>
<td>384</td>
<td>4</td>
<td>1024</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>1342</td>
<td>177</td>
<td>280</td>
<td>5</td>
<td>83</td>
<td>886</td>
<td>080</td>
<td>5</td>
<td>5</td>
<td>2</td>
<td>42</td>
<td>880</td>
<td>5</td>
<td>327</td>
<td>680</td>
<td>5</td>
<td>20</td>
<td>480</td>
<td>5</td>
<td>1280</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>1610</td>
<td>612</td>
<td>736</td>
<td>6</td>
<td>100</td>
<td>663</td>
<td>296</td>
<td>6</td>
<td>6</td>
<td>2</td>
<td>91</td>
<td>456</td>
<td>6</td>
<td>393</td>
<td>216</td>
<td>6</td>
<td>24</td>
<td>576</td>
<td>6</td>
<td>1536</td>
<td>6</td>
</tr>
<tr>
<td>7</td>
<td>1897</td>
<td>048</td>
<td>192</td>
<td>7</td>
<td>117</td>
<td>440</td>
<td>512</td>
<td>7</td>
<td>7</td>
<td>3</td>
<td>40</td>
<td>032</td>
<td>7</td>
<td>458</td>
<td>752</td>
<td>7</td>
<td>28</td>
<td>672</td>
<td>7</td>
<td>1792</td>
<td>7</td>
</tr>
<tr>
<td>8</td>
<td>2147</td>
<td>483</td>
<td>643</td>
<td>8</td>
<td>134</td>
<td>217</td>
<td>728</td>
<td>8</td>
<td>8</td>
<td>3</td>
<td>88</td>
<td>060</td>
<td>8</td>
<td>524</td>
<td>288</td>
<td>8</td>
<td>32</td>
<td>768</td>
<td>8</td>
<td>2048</td>
<td>8</td>
</tr>
<tr>
<td>9</td>
<td>2415</td>
<td>919</td>
<td>104</td>
<td>9</td>
<td>150</td>
<td>994</td>
<td>944</td>
<td>9</td>
<td>9</td>
<td>4</td>
<td>37</td>
<td>184</td>
<td>9</td>
<td>589</td>
<td>824</td>
<td>9</td>
<td>36</td>
<td>864</td>
<td>9</td>
<td>2304</td>
<td>9</td>
</tr>
<tr>
<td>A</td>
<td>2684</td>
<td>354</td>
<td>560</td>
<td>A</td>
<td>167</td>
<td>772</td>
<td>160</td>
<td>A</td>
<td>10</td>
<td>485</td>
<td>760</td>
<td>A</td>
<td>655</td>
<td>360</td>
<td>A</td>
<td>40</td>
<td>960</td>
<td>A</td>
<td>2560</td>
<td>A</td>
<td>160</td>
</tr>
<tr>
<td>B</td>
<td>2952</td>
<td>790</td>
<td>016</td>
<td>B</td>
<td>184</td>
<td>549</td>
<td>376</td>
<td>B</td>
<td>11</td>
<td>534</td>
<td>336</td>
<td>B</td>
<td>720</td>
<td>896</td>
<td>B</td>
<td>45</td>
<td>056</td>
<td>B</td>
<td>2816</td>
<td>B</td>
<td>176</td>
</tr>
<tr>
<td>C</td>
<td>3221</td>
<td>225</td>
<td>472</td>
<td>C</td>
<td>201</td>
<td>326</td>
<td>592</td>
<td>C</td>
<td>12</td>
<td>582</td>
<td>912</td>
<td>C</td>
<td>786</td>
<td>432</td>
<td>C</td>
<td>49</td>
<td>152</td>
<td>C</td>
<td>3072</td>
<td>C</td>
<td>192</td>
</tr>
<tr>
<td>D</td>
<td>3489</td>
<td>660</td>
<td>928</td>
<td>D</td>
<td>218</td>
<td>103</td>
<td>808</td>
<td>D</td>
<td>13</td>
<td>631</td>
<td>486</td>
<td>D</td>
<td>851</td>
<td>968</td>
<td>D</td>
<td>53</td>
<td>248</td>
<td>D</td>
<td>3328</td>
<td>D</td>
<td>208</td>
</tr>
<tr>
<td>E</td>
<td>3758</td>
<td>096</td>
<td>384</td>
<td>E</td>
<td>234</td>
<td>881</td>
<td>024</td>
<td>E</td>
<td>14</td>
<td>680</td>
<td>064</td>
<td>E</td>
<td>917</td>
<td>504</td>
<td>E</td>
<td>57</td>
<td>344</td>
<td>E</td>
<td>3584</td>
<td>E</td>
<td>224</td>
</tr>
<tr>
<td>F</td>
<td>4026</td>
<td>531</td>
<td>840</td>
<td>F</td>
<td>251</td>
<td>658</td>
<td>240</td>
<td>F</td>
<td>15</td>
<td>728</td>
<td>640</td>
<td>F</td>
<td>983</td>
<td>040</td>
<td>F</td>
<td>61</td>
<td>440</td>
<td>F</td>
<td>3840</td>
<td>F</td>
<td>240</td>
</tr>
</tbody>
</table>
### Table A-2  HEXADECIMAL ADDITION

<p>| | | | | | | | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>00</td>
<td>01</td>
<td>02</td>
<td>03</td>
<td>04</td>
<td>05</td>
<td>06</td>
<td>07</td>
<td>08</td>
<td>09</td>
<td>A0</td>
<td>A1</td>
<td>A2</td>
<td>A3</td>
<td>A4</td>
</tr>
<tr>
<td>1</td>
<td>01</td>
<td>02</td>
<td>03</td>
<td>04</td>
<td>05</td>
<td>06</td>
<td>07</td>
<td>08</td>
<td>09</td>
<td>A0</td>
<td>A1</td>
<td>A2</td>
<td>A3</td>
<td>A4</td>
<td>A5</td>
</tr>
<tr>
<td>2</td>
<td>02</td>
<td>03</td>
<td>04</td>
<td>05</td>
<td>06</td>
<td>07</td>
<td>08</td>
<td>09</td>
<td>A0</td>
<td>A1</td>
<td>A2</td>
<td>A3</td>
<td>A4</td>
<td>A5</td>
<td>A6</td>
</tr>
<tr>
<td>3</td>
<td>03</td>
<td>04</td>
<td>05</td>
<td>06</td>
<td>07</td>
<td>08</td>
<td>09</td>
<td>A0</td>
<td>A1</td>
<td>A2</td>
<td>A3</td>
<td>A4</td>
<td>A5</td>
<td>A6</td>
<td>A7</td>
</tr>
<tr>
<td>4</td>
<td>04</td>
<td>05</td>
<td>06</td>
<td>07</td>
<td>08</td>
<td>09</td>
<td>A0</td>
<td>A1</td>
<td>A2</td>
<td>A3</td>
<td>A4</td>
<td>A5</td>
<td>A6</td>
<td>A7</td>
<td>A8</td>
</tr>
<tr>
<td>5</td>
<td>05</td>
<td>06</td>
<td>07</td>
<td>08</td>
<td>09</td>
<td>A0</td>
<td>A1</td>
<td>A2</td>
<td>A3</td>
<td>A4</td>
<td>A5</td>
<td>A6</td>
<td>A7</td>
<td>A8</td>
<td>A9</td>
</tr>
<tr>
<td>6</td>
<td>06</td>
<td>07</td>
<td>08</td>
<td>09</td>
<td>A0</td>
<td>A1</td>
<td>A2</td>
<td>A3</td>
<td>A4</td>
<td>A5</td>
<td>A6</td>
<td>A7</td>
<td>A8</td>
<td>A9</td>
<td>A0</td>
</tr>
<tr>
<td>7</td>
<td>07</td>
<td>08</td>
<td>09</td>
<td>A0</td>
<td>A1</td>
<td>A2</td>
<td>A3</td>
<td>A4</td>
<td>A5</td>
<td>A6</td>
<td>A7</td>
<td>A8</td>
<td>A9</td>
<td>A0</td>
<td>A1</td>
</tr>
<tr>
<td>8</td>
<td>08</td>
<td>09</td>
<td>A0</td>
<td>A1</td>
<td>A2</td>
<td>A3</td>
<td>A4</td>
<td>A5</td>
<td>A6</td>
<td>A7</td>
<td>A8</td>
<td>A9</td>
<td>A0</td>
<td>A1</td>
<td>A2</td>
</tr>
<tr>
<td>9</td>
<td>09</td>
<td>A0</td>
<td>A1</td>
<td>A2</td>
<td>A3</td>
<td>A4</td>
<td>A5</td>
<td>A6</td>
<td>A7</td>
<td>A8</td>
<td>A9</td>
<td>A0</td>
<td>A1</td>
<td>A2</td>
<td>A3</td>
</tr>
<tr>
<td>A</td>
<td>A0</td>
<td>A1</td>
<td>A2</td>
<td>A3</td>
<td>A4</td>
<td>A5</td>
<td>A6</td>
<td>A7</td>
<td>A8</td>
<td>A9</td>
<td>A0</td>
<td>A1</td>
<td>A2</td>
<td>A3</td>
<td>A4</td>
</tr>
<tr>
<td>B</td>
<td>B0</td>
<td>B1</td>
<td>B2</td>
<td>B3</td>
<td>B4</td>
<td>B5</td>
<td>B6</td>
<td>B7</td>
<td>B8</td>
<td>B9</td>
<td>B0</td>
<td>B1</td>
<td>B2</td>
<td>B3</td>
<td>B4</td>
</tr>
<tr>
<td>C</td>
<td>C0</td>
<td>C1</td>
<td>C2</td>
<td>C3</td>
<td>C4</td>
<td>C5</td>
<td>C6</td>
<td>C7</td>
<td>C8</td>
<td>C9</td>
<td>C0</td>
<td>C1</td>
<td>C2</td>
<td>C3</td>
<td>C4</td>
</tr>
<tr>
<td>D</td>
<td>D0</td>
<td>D1</td>
<td>D2</td>
<td>D3</td>
<td>D4</td>
<td>D5</td>
<td>D6</td>
<td>D7</td>
<td>D8</td>
<td>D9</td>
<td>D0</td>
<td>D1</td>
<td>D2</td>
<td>D3</td>
<td>D4</td>
</tr>
<tr>
<td>E</td>
<td>E0</td>
<td>E1</td>
<td>E2</td>
<td>E3</td>
<td>E4</td>
<td>E5</td>
<td>E6</td>
<td>E7</td>
<td>E8</td>
<td>E9</td>
<td>E0</td>
<td>E1</td>
<td>E2</td>
<td>E3</td>
<td>E4</td>
</tr>
<tr>
<td>F</td>
<td>F0</td>
<td>F1</td>
<td>F2</td>
<td>F3</td>
<td>F4</td>
<td>F5</td>
<td>F6</td>
<td>F7</td>
<td>F8</td>
<td>F9</td>
<td>F0</td>
<td>F1</td>
<td>F2</td>
<td>F3</td>
<td>F4</td>
</tr>
</tbody>
</table>

### Table A-3  HEXADECIMAL MULTIPLICATION

<p>| | | | | | | | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
</tr>
<tr>
<td>1</td>
<td>01</td>
<td>02</td>
<td>03</td>
<td>04</td>
<td>05</td>
<td>06</td>
<td>07</td>
<td>08</td>
<td>09</td>
<td>A0</td>
<td>A1</td>
<td>A2</td>
<td>A3</td>
<td>A4</td>
<td>A5</td>
</tr>
<tr>
<td>2</td>
<td>02</td>
<td>04</td>
<td>06</td>
<td>08</td>
<td>0A</td>
<td>0C</td>
<td>0E</td>
<td>00</td>
<td>02</td>
<td>04</td>
<td>06</td>
<td>08</td>
<td>0A</td>
<td>0C</td>
<td>0E</td>
</tr>
<tr>
<td>3</td>
<td>03</td>
<td>06</td>
<td>09</td>
<td>0C</td>
<td>0F</td>
<td>12</td>
<td>15</td>
<td>18</td>
<td>1B</td>
<td>21</td>
<td>24</td>
<td>27</td>
<td>30</td>
<td>33</td>
<td>36</td>
</tr>
<tr>
<td>4</td>
<td>04</td>
<td>08</td>
<td>0C</td>
<td>10</td>
<td>14</td>
<td>18</td>
<td>1C</td>
<td>20</td>
<td>24</td>
<td>28</td>
<td>30</td>
<td>34</td>
<td>38</td>
<td>3C</td>
<td>40</td>
</tr>
<tr>
<td>5</td>
<td>05</td>
<td>0A</td>
<td>0F</td>
<td>14</td>
<td>19</td>
<td>1E</td>
<td>23</td>
<td>28</td>
<td>33</td>
<td>38</td>
<td>40</td>
<td>45</td>
<td>4A</td>
<td>50</td>
<td>55</td>
</tr>
<tr>
<td>6</td>
<td>06</td>
<td>0C</td>
<td>12</td>
<td>18</td>
<td>1E</td>
<td>24</td>
<td>2A</td>
<td>30</td>
<td>36</td>
<td>40</td>
<td>4C</td>
<td>52</td>
<td>58</td>
<td>60</td>
<td>64</td>
</tr>
<tr>
<td>7</td>
<td>07</td>
<td>0E</td>
<td>15</td>
<td>1C</td>
<td>23</td>
<td>2A</td>
<td>31</td>
<td>38</td>
<td>46</td>
<td>4D</td>
<td>53</td>
<td>5A</td>
<td>61</td>
<td>68</td>
<td>75</td>
</tr>
<tr>
<td>8</td>
<td>08</td>
<td>10</td>
<td>20</td>
<td>28</td>
<td>30</td>
<td>38</td>
<td>40</td>
<td>48</td>
<td>50</td>
<td>58</td>
<td>60</td>
<td>68</td>
<td>70</td>
<td>78</td>
<td>80</td>
</tr>
<tr>
<td>9</td>
<td>09</td>
<td>12</td>
<td>24</td>
<td>36</td>
<td>3F</td>
<td>48</td>
<td>51</td>
<td>5A</td>
<td>63</td>
<td>6C</td>
<td>75</td>
<td>7E</td>
<td>87</td>
<td>90</td>
<td>A1</td>
</tr>
<tr>
<td>A</td>
<td>0A</td>
<td>14</td>
<td>28</td>
<td>32</td>
<td>3C</td>
<td>46</td>
<td>50</td>
<td>5A</td>
<td>64</td>
<td>68</td>
<td>70</td>
<td>72</td>
<td>76</td>
<td>80</td>
<td>84</td>
</tr>
<tr>
<td>B</td>
<td>0B</td>
<td>16</td>
<td>22</td>
<td>2C</td>
<td>37</td>
<td>42</td>
<td>4D</td>
<td>58</td>
<td>63</td>
<td>6E</td>
<td>79</td>
<td>84</td>
<td>8F</td>
<td>9A</td>
<td>A5</td>
</tr>
<tr>
<td>C</td>
<td>0C</td>
<td>18</td>
<td>24</td>
<td>30</td>
<td>3C</td>
<td>48</td>
<td>54</td>
<td>60</td>
<td>6C</td>
<td>78</td>
<td>84</td>
<td>90</td>
<td>9C</td>
<td>A8</td>
<td>B4</td>
</tr>
<tr>
<td>D</td>
<td>0D</td>
<td>1A</td>
<td>27</td>
<td>34</td>
<td>41</td>
<td>4E</td>
<td>5B</td>
<td>68</td>
<td>75</td>
<td>82</td>
<td>8F</td>
<td>9C</td>
<td>A9</td>
<td>B6</td>
<td>C3</td>
</tr>
<tr>
<td>E</td>
<td>0E</td>
<td>1C</td>
<td>2A</td>
<td>38</td>
<td>46</td>
<td>54</td>
<td>62</td>
<td>70</td>
<td>7E</td>
<td>8C</td>
<td>9A</td>
<td>A8</td>
<td>B6</td>
<td>C4</td>
<td>D2</td>
</tr>
<tr>
<td>F</td>
<td>0F</td>
<td>1E</td>
<td>2D</td>
<td>3C</td>
<td>4B</td>
<td>5A</td>
<td>69</td>
<td>78</td>
<td>87</td>
<td>96</td>
<td>A5</td>
<td>B4</td>
<td>C3</td>
<td>D2</td>
<td>E1</td>
</tr>
</tbody>
</table>
### ASCII CHARACTER SET AND HEX-ASCII CONVERSION

Table A-4 is a table representing the ASCII character set.

#### Table A-4  ASCII CHARACTER SET

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>NUL</td>
<td>DLE</td>
<td>SP</td>
<td>@</td>
<td>P</td>
<td>'</td>
<td>p</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>SOH</td>
<td>DC1</td>
<td>!</td>
<td>A</td>
<td>Q</td>
<td>a</td>
<td>q</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>STX</td>
<td>DC2</td>
<td>&quot;</td>
<td>B</td>
<td>R</td>
<td>b</td>
<td>r</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>ETX</td>
<td>DC3</td>
<td>#</td>
<td>C</td>
<td>S</td>
<td>c</td>
<td>s</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>EOT</td>
<td>DC4</td>
<td>$</td>
<td>D</td>
<td>T</td>
<td>d</td>
<td>t</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>ENQ</td>
<td>NAK</td>
<td>%</td>
<td>E</td>
<td>U</td>
<td>e</td>
<td>u</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>ACK</td>
<td>SYN</td>
<td>&amp;</td>
<td>F</td>
<td>V</td>
<td>f</td>
<td>v</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>BEL</td>
<td>ETB</td>
<td>'</td>
<td>G</td>
<td>W</td>
<td>g</td>
<td>w</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>BS</td>
<td>CAN</td>
<td>(</td>
<td>H</td>
<td>X</td>
<td>h</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>HT</td>
<td>EM</td>
<td>)</td>
<td>I</td>
<td>Y</td>
<td>i</td>
<td>y</td>
<td></td>
</tr>
<tr>
<td>A</td>
<td>LF</td>
<td>SUB</td>
<td>*</td>
<td>J</td>
<td>Z</td>
<td>j</td>
<td>z</td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>VT</td>
<td>ESC</td>
<td>+</td>
<td>K</td>
<td>[</td>
<td>k</td>
<td>{</td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>FF</td>
<td>FS</td>
<td>,</td>
<td>L</td>
<td>\</td>
<td>l</td>
<td></td>
<td></td>
</tr>
<tr>
<td>D</td>
<td>CR</td>
<td>GS</td>
<td>=</td>
<td>M</td>
<td>]</td>
<td>m</td>
<td>}</td>
<td></td>
</tr>
<tr>
<td>E</td>
<td>SO</td>
<td>RS</td>
<td>&gt;</td>
<td>N</td>
<td>‡</td>
<td>n</td>
<td>~</td>
<td></td>
</tr>
<tr>
<td>F</td>
<td>SI</td>
<td>US</td>
<td>/</td>
<td>O</td>
<td>—</td>
<td>o</td>
<td>DEL</td>
<td></td>
</tr>
</tbody>
</table>

- **NUL**: Null
- **SOH**: Start of Heading
- **STX**: Start of Text
- **ETX**: End of Text
- **EOT**: End of Transmission
- **ENQ**: Enquiry
- **ACK**: Acknowledge
- **BEL**: Bell
- **BS**: Backspace
- **HT**: Horizontal Tabulation
- **LF**: Line Feed
- **VT**: Vertical Tabulation
- **FF**: Form Feed
- **CR**: Carriage Return
- **SO**: Shift Out
- **SI**: Shift In
- **SP**: Space
- **DLE**: Data Link Escape
- **DC1**: Device Control 1
- **DC2**: Device Control 2
- **DC3**: Device Control 3
- **DC4**: Device Control 4
- **NAK**: Negative Acknowledge
- **SYN**: Synchronous Idle
- **ETB**: End of Transmission Block
- **CAN**: Cancel
- **EM**: End of Medium
- **SUB**: Substitute
- **ESC**: ESCAPE
- **FS**: File Separator
- **GS**: Group Separator
- **RS**: Record Separator
- **US**: Unit Separator
- **DEL**: Delete
Appendix A

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**

<table>
<thead>
<tr>
<th>$2^{**n}$</th>
<th>$n$</th>
</tr>
</thead>
<tbody>
<tr>
<td>256</td>
<td>8</td>
</tr>
<tr>
<td>512</td>
<td>9</td>
</tr>
<tr>
<td>1024</td>
<td>10</td>
</tr>
<tr>
<td>2048</td>
<td>11</td>
</tr>
<tr>
<td>4096</td>
<td>12</td>
</tr>
<tr>
<td>8192</td>
<td>13</td>
</tr>
<tr>
<td>16384</td>
<td>14</td>
</tr>
<tr>
<td>32768</td>
<td>15</td>
</tr>
<tr>
<td>65536</td>
<td>16</td>
</tr>
<tr>
<td>131072</td>
<td>17</td>
</tr>
<tr>
<td>262144</td>
<td>18</td>
</tr>
<tr>
<td>524288</td>
<td>19</td>
</tr>
<tr>
<td>1048576</td>
<td>20</td>
</tr>
<tr>
<td>2097152</td>
<td>21</td>
</tr>
<tr>
<td>4194304</td>
<td>22</td>
</tr>
<tr>
<td>8388608</td>
<td>23</td>
</tr>
<tr>
<td>16777216</td>
<td>24</td>
</tr>
</tbody>
</table>

**Powers of 16**

<table>
<thead>
<tr>
<th>$16^{**n}$</th>
<th>$n$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>16</td>
<td>1</td>
</tr>
<tr>
<td>256</td>
<td>2</td>
</tr>
<tr>
<td>4096</td>
<td>3</td>
</tr>
<tr>
<td>65536</td>
<td>4</td>
</tr>
<tr>
<td>1048576</td>
<td>5</td>
</tr>
<tr>
<td>16777216</td>
<td>6</td>
</tr>
<tr>
<td>268435456</td>
<td>7</td>
</tr>
<tr>
<td>4294967296</td>
<td>8</td>
</tr>
<tr>
<td>68719476736</td>
<td>9</td>
</tr>
<tr>
<td>1099511627776</td>
<td>10</td>
</tr>
<tr>
<td>17592186044416</td>
<td>11</td>
</tr>
<tr>
<td>281474976710656</td>
<td>12</td>
</tr>
<tr>
<td>4503599627370496</td>
<td>13</td>
</tr>
<tr>
<td>72057594037927936</td>
<td>14</td>
</tr>
<tr>
<td>1152921504606846976</td>
<td>15</td>
</tr>
</tbody>
</table>
### APPENDIX B
### INSTRUCTION INDEX

<table>
<thead>
<tr>
<th>MNEMONIC LISTING</th>
<th>OPCODE</th>
<th>PAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>MNEMONIC</strong></td>
<td><strong>INSTRUCTION</strong></td>
<td></td>
</tr>
<tr>
<td>ACBB</td>
<td>Add compare and branch byte</td>
<td>9D 214</td>
</tr>
<tr>
<td>ACBD</td>
<td>Add compare and branch double</td>
<td>6F 214</td>
</tr>
<tr>
<td>ACBF</td>
<td>Add compare and branch floating</td>
<td>4F 214</td>
</tr>
<tr>
<td>ACBL</td>
<td>Add compare and branch long</td>
<td>F1 214</td>
</tr>
<tr>
<td>ACBW</td>
<td>Add compare and branch word</td>
<td>3D 214</td>
</tr>
<tr>
<td>ADAWI</td>
<td>Add aligned word interlocked</td>
<td>58 143</td>
</tr>
<tr>
<td>ADDB2</td>
<td>Add byte 2 operand</td>
<td>80 140</td>
</tr>
<tr>
<td>ADDB3</td>
<td>Add byte 3 operand</td>
<td>81 140</td>
</tr>
<tr>
<td>ADDD2</td>
<td>Add double 2 operand</td>
<td>60 140</td>
</tr>
<tr>
<td>ADDD3</td>
<td>Add double 3 operand</td>
<td>61 140</td>
</tr>
<tr>
<td>ADDF2</td>
<td>Add floating 2 operand</td>
<td>40 140</td>
</tr>
<tr>
<td>ADDF3</td>
<td>Add floating 3 operand</td>
<td>41 140</td>
</tr>
<tr>
<td>ADDL2</td>
<td>Add long 2 operand</td>
<td>C0 140</td>
</tr>
<tr>
<td>ADDL3</td>
<td>Add long 3 operand</td>
<td>C1 140</td>
</tr>
<tr>
<td>ADDP4</td>
<td>Add packed 4 operand</td>
<td>20 256</td>
</tr>
<tr>
<td>ADDP6</td>
<td>Add packed 6 operand</td>
<td>21 256</td>
</tr>
<tr>
<td>ADDW2</td>
<td>Add word 2 operand</td>
<td>A0 140</td>
</tr>
<tr>
<td>ADDW3</td>
<td>Add word 3 operand</td>
<td>A1 140</td>
</tr>
<tr>
<td>ADWC</td>
<td>Add with carry</td>
<td>D8 142</td>
</tr>
<tr>
<td>AOBLEQ</td>
<td>Add one and branch on less or equal</td>
<td>F3 216</td>
</tr>
<tr>
<td>AOBLS</td>
<td>Add one and branch on less</td>
<td>F2 216</td>
</tr>
<tr>
<td>ASHL</td>
<td>Arithmetic shift long</td>
<td>78 159</td>
</tr>
<tr>
<td>ASHP</td>
<td>Arithmetic shift and round packed</td>
<td>F8 272</td>
</tr>
<tr>
<td>ASHQ</td>
<td>Arithmetic shift quad</td>
<td>79 159</td>
</tr>
<tr>
<td>BBC</td>
<td>Branch on bit clear</td>
<td>E1 209</td>
</tr>
<tr>
<td>BBCC</td>
<td>Branch on bit clear and clear</td>
<td>E5 210</td>
</tr>
<tr>
<td>BBCCI</td>
<td>Branch on bit clear and clear interlocked</td>
<td>E7 211</td>
</tr>
<tr>
<td>BBCS</td>
<td>Branch on bit clear and set</td>
<td>E3 210</td>
</tr>
<tr>
<td>BBS</td>
<td>Branch on bit set</td>
<td>E0 209</td>
</tr>
<tr>
<td>BBSC</td>
<td>Branch on bit set and clear</td>
<td>E4 210</td>
</tr>
<tr>
<td>BBSS</td>
<td>Branch on bit set and set</td>
<td>E2 210</td>
</tr>
<tr>
<td>BBSSI</td>
<td>Branch on bit set and set interlocked</td>
<td>E6 211</td>
</tr>
<tr>
<td>MNEMONIC</td>
<td>INSTRUCTION</td>
<td>OPCODE</td>
</tr>
<tr>
<td>----------</td>
<td>-------------------------------------------</td>
<td>--------</td>
</tr>
<tr>
<td>BCC</td>
<td>Branch on carry clear</td>
<td>1E</td>
</tr>
<tr>
<td>BCS</td>
<td>Branch on carry set</td>
<td>1F</td>
</tr>
<tr>
<td>BEQL</td>
<td>Branch on equal (signed)</td>
<td>13</td>
</tr>
<tr>
<td>BEQU</td>
<td>Branch on equal unsigned</td>
<td>13</td>
</tr>
<tr>
<td>BGEQ</td>
<td>Branch on greater or equal</td>
<td>18</td>
</tr>
<tr>
<td>BGEU</td>
<td>Branch on greater or equal unsigned</td>
<td>1E</td>
</tr>
<tr>
<td>BGTR</td>
<td>Branch on greater</td>
<td>14</td>
</tr>
<tr>
<td>BGTRU</td>
<td>Branch on greater unsigned</td>
<td>1A</td>
</tr>
<tr>
<td>BICB2</td>
<td>Bit clear byte 2 operand</td>
<td>8A</td>
</tr>
<tr>
<td>BICB3</td>
<td>Bit clear byte 3 operand</td>
<td>8B</td>
</tr>
<tr>
<td>BICL2</td>
<td>Bit clear long 2 operand</td>
<td>CA</td>
</tr>
<tr>
<td>BICL3</td>
<td>Bit clear long 3 operand</td>
<td>CB</td>
</tr>
<tr>
<td>BICPSW</td>
<td>Bit clear program status word</td>
<td>B9</td>
</tr>
<tr>
<td>BICW2</td>
<td>Bit clear word 2 operand</td>
<td>AA</td>
</tr>
<tr>
<td>BICW3</td>
<td>Bit clear word 3 operand</td>
<td>AB</td>
</tr>
<tr>
<td>BISB2</td>
<td>Bit set byte 2 operand</td>
<td>88</td>
</tr>
<tr>
<td>BISB3</td>
<td>Bit set byte 3 operand</td>
<td>89</td>
</tr>
<tr>
<td>BISL2</td>
<td>Bit set long 2 operand</td>
<td>C8</td>
</tr>
<tr>
<td>BISL3</td>
<td>Bit set long 3 operand</td>
<td>C9</td>
</tr>
<tr>
<td>BISPSW</td>
<td>Bit set program status word</td>
<td>B8</td>
</tr>
<tr>
<td>BISW2</td>
<td>Bit set word 2 operand</td>
<td>A8</td>
</tr>
<tr>
<td>BISW3</td>
<td>Bit set word 3 operand</td>
<td>A9</td>
</tr>
<tr>
<td>BITB</td>
<td>Bit test byte</td>
<td>93</td>
</tr>
<tr>
<td>BITL</td>
<td>Bit test long</td>
<td>D3</td>
</tr>
<tr>
<td>BITW</td>
<td>Bit test word</td>
<td>B3</td>
</tr>
<tr>
<td>BLBC</td>
<td>Branch on low bit clear</td>
<td>E9</td>
</tr>
<tr>
<td>BLBS</td>
<td>Branch on low bit set</td>
<td>E8</td>
</tr>
<tr>
<td>BLEQ</td>
<td>Branch on less or equal</td>
<td>15</td>
</tr>
<tr>
<td>BLEQU</td>
<td>Branch on less or equal unsigned</td>
<td>1B</td>
</tr>
<tr>
<td>BLSS</td>
<td>Branch on less</td>
<td>19</td>
</tr>
<tr>
<td>BLSSU</td>
<td>Branch on less unsigned</td>
<td>1F</td>
</tr>
<tr>
<td>BNEQ</td>
<td>Branch on not equal</td>
<td>12</td>
</tr>
<tr>
<td>BNEQU</td>
<td>Branch on not equal unsigned</td>
<td>12</td>
</tr>
<tr>
<td>BPT</td>
<td>Break point fault</td>
<td>03</td>
</tr>
<tr>
<td>BRB</td>
<td>Branch with byte displacement</td>
<td>11</td>
</tr>
<tr>
<td>BRW</td>
<td>Branch with word displacement</td>
<td>31</td>
</tr>
<tr>
<td>MNEMONIC</td>
<td>INSTRUCTION</td>
<td>OPCODE PAGE</td>
</tr>
<tr>
<td>----------</td>
<td>--------------------------------------------------</td>
<td>-------------</td>
</tr>
<tr>
<td>BSBB</td>
<td>Branch to subroutine with byte displacement</td>
<td>10 220</td>
</tr>
<tr>
<td>BSBW</td>
<td>Branch to subroutine with word displacement</td>
<td>30 220</td>
</tr>
<tr>
<td>BVC</td>
<td>Branch on overflow clear</td>
<td>1C 206</td>
</tr>
<tr>
<td>BVS</td>
<td>Branch on overflow set</td>
<td>1D 206</td>
</tr>
<tr>
<td>CALLG</td>
<td>Call with general argument list</td>
<td>FA 226</td>
</tr>
<tr>
<td>CALLS</td>
<td>Call with stack</td>
<td>FB 228</td>
</tr>
<tr>
<td>CASEB</td>
<td>Case byte</td>
<td>8F 218</td>
</tr>
<tr>
<td>CASEL</td>
<td>Case long</td>
<td>CF 218</td>
</tr>
<tr>
<td>CASEW</td>
<td>Case word</td>
<td>AF 218</td>
</tr>
<tr>
<td>CHME</td>
<td>Change mode to executive</td>
<td>BD —</td>
</tr>
<tr>
<td>CHMK</td>
<td>Change mode to kernel</td>
<td>BC —</td>
</tr>
<tr>
<td>CHMS</td>
<td>Change mode to supervisor</td>
<td>BE —</td>
</tr>
<tr>
<td>CHMU</td>
<td>Change mode to user</td>
<td>BF —</td>
</tr>
<tr>
<td>CLRB</td>
<td>Clear byte</td>
<td>94 131</td>
</tr>
<tr>
<td>CLRD</td>
<td>Clear double</td>
<td>7C 131</td>
</tr>
<tr>
<td>CLRF</td>
<td>Clear float</td>
<td>D4 131</td>
</tr>
<tr>
<td>CLRRL</td>
<td>Clear long</td>
<td>D4 131</td>
</tr>
<tr>
<td>CLRQ</td>
<td>Clear quad</td>
<td>7C 131</td>
</tr>
<tr>
<td>CLRW</td>
<td>Clear word</td>
<td>B4 131</td>
</tr>
<tr>
<td>CMPB</td>
<td>Compare byte</td>
<td>91 137</td>
</tr>
<tr>
<td>CMPC3</td>
<td>Compare character 3 operand</td>
<td>29 240</td>
</tr>
<tr>
<td>CMPC5</td>
<td>Compare character 5 operand</td>
<td>2D 240</td>
</tr>
<tr>
<td>CMPD</td>
<td>Compare double</td>
<td>71 137</td>
</tr>
<tr>
<td>CMPF</td>
<td>Compare floating</td>
<td>51 137</td>
</tr>
<tr>
<td>CMPL</td>
<td>Compare long</td>
<td>D1 137</td>
</tr>
<tr>
<td>CMPP3</td>
<td>Compare packed 3 operand</td>
<td>35 255</td>
</tr>
<tr>
<td>CMPP4</td>
<td>Compare packed 4 operand</td>
<td>37 255</td>
</tr>
<tr>
<td>CMPV</td>
<td>Compare field</td>
<td>EC 201</td>
</tr>
<tr>
<td>CMPW</td>
<td>Compare word</td>
<td>B1 137</td>
</tr>
<tr>
<td>CMPZV</td>
<td>Compare zero-extended field</td>
<td>ED 201</td>
</tr>
<tr>
<td>CRC</td>
<td>Calculate cyclic redundancy check</td>
<td>0B 246</td>
</tr>
<tr>
<td>CVTBD</td>
<td>Convert byte to double</td>
<td>6C 134</td>
</tr>
<tr>
<td>CVTBF</td>
<td>Convert byte to float</td>
<td>4C 134</td>
</tr>
<tr>
<td>CVTBL</td>
<td>Convert byte to long</td>
<td>98 134</td>
</tr>
<tr>
<td>CVTBW</td>
<td>Convert byte to word</td>
<td>99 134</td>
</tr>
<tr>
<td>CVTDB</td>
<td>Convert double to byte</td>
<td>68 134</td>
</tr>
<tr>
<td>MNEMONIC</td>
<td>INSTRUCTION</td>
<td>OPCODE PAGE</td>
</tr>
<tr>
<td>----------</td>
<td>--------------------------------------------------</td>
<td>-------------</td>
</tr>
<tr>
<td>CVTDF</td>
<td>Convert double to float</td>
<td>76 134</td>
</tr>
<tr>
<td>CVTDL</td>
<td>Convert double to long</td>
<td>6A 134</td>
</tr>
<tr>
<td>CVTDW</td>
<td>Convert double to word</td>
<td>69 134</td>
</tr>
<tr>
<td>CVTFB</td>
<td>Convert float to byte</td>
<td>48 134</td>
</tr>
<tr>
<td>CVTFD</td>
<td>Convert float to double</td>
<td>56 134</td>
</tr>
<tr>
<td>CVTFL</td>
<td>Convert float to long</td>
<td>4A 134</td>
</tr>
<tr>
<td>CVTFW</td>
<td>Convert float to word</td>
<td>49 134</td>
</tr>
<tr>
<td>CVTLB</td>
<td>Convert long to byte</td>
<td>F6 134</td>
</tr>
<tr>
<td>CVTLD</td>
<td>Convert long to double</td>
<td>6E 134</td>
</tr>
<tr>
<td>CVTLF</td>
<td>Convert long to float</td>
<td>4E 134</td>
</tr>
<tr>
<td>CVTPLP</td>
<td>Convert long to packed</td>
<td>F9 263</td>
</tr>
<tr>
<td>CVTLW</td>
<td>Convert long to word</td>
<td>F7 134</td>
</tr>
<tr>
<td>CVTPL</td>
<td>Convert packed to long</td>
<td>36 264</td>
</tr>
<tr>
<td>CVTTP</td>
<td>Convert trailing numeric to packed</td>
<td>26 267</td>
</tr>
<tr>
<td>CVTPT</td>
<td>Convert packed to trailing numeric</td>
<td>24 265</td>
</tr>
<tr>
<td>CVTPS</td>
<td>Convert packed to leading separate numeric</td>
<td>08 269</td>
</tr>
<tr>
<td>CVTRDL</td>
<td>Convert rounded double to long</td>
<td>6B 134</td>
</tr>
<tr>
<td>CVTRFL</td>
<td>Convert rounded float to long</td>
<td>4B 134</td>
</tr>
<tr>
<td>CVTSP</td>
<td>Convert leading separate numeric to packed</td>
<td>09 271</td>
</tr>
<tr>
<td>CVTWB</td>
<td>Convert word to byte</td>
<td>33 134</td>
</tr>
<tr>
<td>CVTWD</td>
<td>Convert word to double</td>
<td>6D 134</td>
</tr>
<tr>
<td>CVTWF</td>
<td>Convert word to float</td>
<td>4D 134</td>
</tr>
<tr>
<td>CVTWL</td>
<td>Convert word to long</td>
<td>32 134</td>
</tr>
<tr>
<td>DECB</td>
<td>Decrement byte</td>
<td>97 146</td>
</tr>
<tr>
<td>DECL</td>
<td>Decrement long</td>
<td>D7 146</td>
</tr>
<tr>
<td>DECW</td>
<td>Decrement word</td>
<td>B7 146</td>
</tr>
<tr>
<td>DIVB2</td>
<td>Divide byte 2 operand</td>
<td>86 152</td>
</tr>
<tr>
<td>DIVB3</td>
<td>Divide byte 3 operand</td>
<td>87 152</td>
</tr>
<tr>
<td>DIVD2</td>
<td>Divide double 2 operand</td>
<td>66 152</td>
</tr>
<tr>
<td>DIVD3</td>
<td>Divide double 3 operand</td>
<td>67 152</td>
</tr>
<tr>
<td>DIVF2</td>
<td>Divide floating 2 operand</td>
<td>46 152</td>
</tr>
<tr>
<td>DIVF3</td>
<td>Divide floating 3 operand</td>
<td>47 152</td>
</tr>
<tr>
<td>DIVL2</td>
<td>Divide long 2 operand</td>
<td>C6 152</td>
</tr>
<tr>
<td>DIVL3</td>
<td>Divide long 3 operand</td>
<td>C7 152</td>
</tr>
<tr>
<td>DIVP</td>
<td>Divide packed</td>
<td>27 261</td>
</tr>
<tr>
<td>DIVW2</td>
<td>Divide word 2 operand</td>
<td>A6 152</td>
</tr>
<tr>
<td>DIVW3</td>
<td>Divide word 3 operand</td>
<td>A7 152</td>
</tr>
</tbody>
</table>

336
<table>
<thead>
<tr>
<th>MNEMONIC</th>
<th>INSTRUCTION</th>
<th>Opcode</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>EDITPC</td>
<td>Edit packed to character</td>
<td>38</td>
<td>277</td>
</tr>
<tr>
<td>EDIV</td>
<td>Extended divide</td>
<td>7B</td>
<td>154</td>
</tr>
<tr>
<td>EMODD</td>
<td>Extended modulus double</td>
<td>74</td>
<td>151</td>
</tr>
<tr>
<td>EMODF</td>
<td>Extended modulus floating</td>
<td>54</td>
<td>151</td>
</tr>
<tr>
<td>EMUL</td>
<td>Extended multiply</td>
<td>7A</td>
<td>150</td>
</tr>
<tr>
<td>EXTV</td>
<td>Extract field</td>
<td>EE</td>
<td>199</td>
</tr>
<tr>
<td>EXTZV</td>
<td>Extract zero-extended field</td>
<td>EF</td>
<td>199</td>
</tr>
<tr>
<td>FFC</td>
<td>Find first clear bit</td>
<td>EB</td>
<td>197</td>
</tr>
<tr>
<td>FFS</td>
<td>Find first set bit</td>
<td>EA</td>
<td>197</td>
</tr>
<tr>
<td>HALT</td>
<td>Halt</td>
<td>00</td>
<td>314</td>
</tr>
<tr>
<td>INCB</td>
<td>Increment byte</td>
<td>96</td>
<td>138</td>
</tr>
<tr>
<td>INCL</td>
<td>Increment long</td>
<td>D6</td>
<td>138</td>
</tr>
<tr>
<td>INCW</td>
<td>Increment word</td>
<td>B6</td>
<td>138</td>
</tr>
<tr>
<td>INDEX</td>
<td>Compute index</td>
<td>0A</td>
<td>174</td>
</tr>
<tr>
<td>INSQH</td>
<td>Insert into queue head, interlocked</td>
<td>5C</td>
<td>188</td>
</tr>
<tr>
<td>INSQTI</td>
<td>Insert into queue tail, interlocked</td>
<td>5D</td>
<td>190</td>
</tr>
<tr>
<td>INSQUE</td>
<td>Insert into queue</td>
<td>0E</td>
<td>180</td>
</tr>
<tr>
<td>INSV</td>
<td>Insert field</td>
<td>F0</td>
<td>203</td>
</tr>
<tr>
<td>JMP</td>
<td>Jump</td>
<td>17</td>
<td>208</td>
</tr>
<tr>
<td>JSB</td>
<td>Jump to subroutine</td>
<td>16</td>
<td>220</td>
</tr>
<tr>
<td>LDPCTX</td>
<td>Load process context</td>
<td>16</td>
<td></td>
</tr>
<tr>
<td>LOCC</td>
<td>Locate character</td>
<td>3A</td>
<td>243</td>
</tr>
<tr>
<td>MATCHC</td>
<td>Match characters</td>
<td>39</td>
<td>244</td>
</tr>
<tr>
<td>MCOMB</td>
<td>Move complemented byte</td>
<td>92</td>
<td>133</td>
</tr>
<tr>
<td>MCOML</td>
<td>Move complemented long</td>
<td>D2</td>
<td>133</td>
</tr>
<tr>
<td>MCOMW</td>
<td>Move complemented word</td>
<td>B2</td>
<td>133</td>
</tr>
<tr>
<td>MFPR</td>
<td>Move from privilege register</td>
<td>DB</td>
<td></td>
</tr>
<tr>
<td>MNEGB</td>
<td>Move negated byte</td>
<td>8E</td>
<td>132</td>
</tr>
<tr>
<td>MNEG</td>
<td>Move negated double</td>
<td>72</td>
<td>132</td>
</tr>
<tr>
<td>MNEGF</td>
<td>Move negated floating</td>
<td>52</td>
<td>132</td>
</tr>
<tr>
<td>MNEG</td>
<td>Move negated long</td>
<td>CE</td>
<td>132</td>
</tr>
<tr>
<td>MNEG</td>
<td>Move negated word</td>
<td>AE</td>
<td>132</td>
</tr>
<tr>
<td>MOVAB</td>
<td>Move address of byte</td>
<td>9E</td>
<td>171</td>
</tr>
<tr>
<td>MOVAD</td>
<td>Move address of double</td>
<td>7E</td>
<td>171</td>
</tr>
<tr>
<td>MOVAF</td>
<td>Move address of float</td>
<td>DE</td>
<td>171</td>
</tr>
<tr>
<td>MOVAL</td>
<td>Move address of long</td>
<td>DE</td>
<td>171</td>
</tr>
<tr>
<td>MNEMONIC</td>
<td>INSTRUCTION</td>
<td>Opcode</td>
<td>Page</td>
</tr>
<tr>
<td>-----------</td>
<td>--------------------------------------</td>
<td>--------</td>
<td>------</td>
</tr>
<tr>
<td>MOVAQ</td>
<td>Move address of quad</td>
<td>7E</td>
<td>171</td>
</tr>
<tr>
<td>MOVAW</td>
<td>Move address of word</td>
<td>3E</td>
<td>171</td>
</tr>
<tr>
<td>MOVB</td>
<td>Move byte</td>
<td>90</td>
<td>129</td>
</tr>
<tr>
<td>MOVC3</td>
<td>Move character 3 operand</td>
<td>28</td>
<td>234</td>
</tr>
<tr>
<td>MOVC5</td>
<td>Move character 5 operand</td>
<td>2C</td>
<td>234</td>
</tr>
<tr>
<td>MOVD</td>
<td>Move double</td>
<td>70</td>
<td>129</td>
</tr>
<tr>
<td>MOVF</td>
<td>Move float</td>
<td>50</td>
<td>129</td>
</tr>
<tr>
<td>MOVL</td>
<td>Move long</td>
<td>D0</td>
<td>129</td>
</tr>
<tr>
<td>MOVP</td>
<td>Move packed</td>
<td>34</td>
<td>254</td>
</tr>
<tr>
<td>MOVPSL</td>
<td>Move processor status longword</td>
<td>DC</td>
<td>169</td>
</tr>
<tr>
<td>MOVQ</td>
<td>Move quad</td>
<td>7D</td>
<td>129</td>
</tr>
<tr>
<td>MOVTC</td>
<td>Move translated characters</td>
<td>2E</td>
<td>236</td>
</tr>
<tr>
<td>MOVTUC</td>
<td>Move translated until character</td>
<td>2F</td>
<td>238</td>
</tr>
<tr>
<td>MOVW</td>
<td>Move word</td>
<td>B0</td>
<td>129</td>
</tr>
<tr>
<td>MOVZBL</td>
<td>Move zero-extended byte to long</td>
<td>9A</td>
<td>136</td>
</tr>
<tr>
<td>MOVZBW</td>
<td>Move zero-extended byte to word</td>
<td>9B</td>
<td>136</td>
</tr>
<tr>
<td>MOVZWL</td>
<td>Move zero-extended word to long</td>
<td>3C</td>
<td>136</td>
</tr>
<tr>
<td>MTPR</td>
<td>Move to privilege register</td>
<td>DA</td>
<td>—</td>
</tr>
<tr>
<td>MULB2</td>
<td>Multiply byte 2 operand</td>
<td>84</td>
<td>148</td>
</tr>
<tr>
<td>MULB3</td>
<td>Multiply byte 3 operand</td>
<td>85</td>
<td>148</td>
</tr>
<tr>
<td>MULD2</td>
<td>Multiply double 2 operand</td>
<td>64</td>
<td>148</td>
</tr>
<tr>
<td>MULD3</td>
<td>Multiply double 3 operand</td>
<td>65</td>
<td>148</td>
</tr>
<tr>
<td>MULF2</td>
<td>Multiply floating 2 operand</td>
<td>44</td>
<td>148</td>
</tr>
<tr>
<td>MULF3</td>
<td>Multiply floating 3 operand</td>
<td>45</td>
<td>148</td>
</tr>
<tr>
<td>MULL2</td>
<td>Multiply long 2 operand</td>
<td>C4</td>
<td>148</td>
</tr>
<tr>
<td>MULL3</td>
<td>Multiply long 3 operand</td>
<td>C5</td>
<td>148</td>
</tr>
<tr>
<td>MULP</td>
<td>Multiply packed</td>
<td>25</td>
<td>260</td>
</tr>
<tr>
<td>MULW2</td>
<td>Multiply word 2 operand</td>
<td>A4</td>
<td>148</td>
</tr>
<tr>
<td>MULW3</td>
<td>Multiply word 3 operand</td>
<td>A5</td>
<td>148</td>
</tr>
<tr>
<td>NOP</td>
<td>No operation</td>
<td>01</td>
<td></td>
</tr>
<tr>
<td>POLYD</td>
<td>Evaluate polynomial double</td>
<td>75</td>
<td>161</td>
</tr>
<tr>
<td>POLYF</td>
<td>Evaluate polynomial floating</td>
<td>55</td>
<td>161</td>
</tr>
<tr>
<td>POPR</td>
<td>Pop registers</td>
<td>BA</td>
<td></td>
</tr>
<tr>
<td>PROBER</td>
<td>Probe read access</td>
<td>0C</td>
<td>—</td>
</tr>
<tr>
<td>PROBEW</td>
<td>Probe write access</td>
<td>0D</td>
<td>—</td>
</tr>
<tr>
<td>PUSHAB</td>
<td>Push address byte</td>
<td>9F</td>
<td>171</td>
</tr>
</tbody>
</table>

338
<table>
<thead>
<tr>
<th>MNEMONIC</th>
<th>INSTRUCTION</th>
<th>Opcode</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>PUSHAD</td>
<td>Push address of double</td>
<td>7F</td>
<td>171</td>
</tr>
<tr>
<td>PUSHAF</td>
<td>Push address of float</td>
<td>DF</td>
<td>171</td>
</tr>
<tr>
<td>PUSHAL</td>
<td>Push address of long</td>
<td>DF</td>
<td>171</td>
</tr>
<tr>
<td>PUSHAQ</td>
<td>Push address of quad</td>
<td>7F</td>
<td>171</td>
</tr>
<tr>
<td>PUSHAW</td>
<td>Push address of word</td>
<td>3F</td>
<td>171</td>
</tr>
<tr>
<td>PUSHL</td>
<td>Push long</td>
<td>DD</td>
<td>130</td>
</tr>
<tr>
<td>PUSHR</td>
<td>Push registers</td>
<td>BB</td>
<td>166</td>
</tr>
<tr>
<td>REI</td>
<td>Return from exception or interrupt</td>
<td>02</td>
<td>311</td>
</tr>
<tr>
<td>REMQHI</td>
<td>Remove from queue head, interlocked</td>
<td>5E</td>
<td>192</td>
</tr>
<tr>
<td>REMQTI</td>
<td>Remove from queue tail, interlocked</td>
<td>5F</td>
<td>194</td>
</tr>
<tr>
<td>REMQUE</td>
<td>Remove from queue</td>
<td>0F</td>
<td>182</td>
</tr>
<tr>
<td>RET</td>
<td>Return from called procedure</td>
<td>04</td>
<td>230</td>
</tr>
<tr>
<td>ROTL</td>
<td>Rotate long</td>
<td>9C</td>
<td>160</td>
</tr>
<tr>
<td>RSB</td>
<td>Return from subroutine</td>
<td>05</td>
<td>221</td>
</tr>
<tr>
<td>SBWC</td>
<td>Subtract with carry</td>
<td>D9</td>
<td>147</td>
</tr>
<tr>
<td>SCANC</td>
<td>Scan for character</td>
<td>2A</td>
<td>242</td>
</tr>
<tr>
<td>SKPC</td>
<td>Skip character</td>
<td>3B</td>
<td>243</td>
</tr>
<tr>
<td>SOBGEQ</td>
<td>Subtract one and branch on greater or equal</td>
<td>F4</td>
<td>217</td>
</tr>
<tr>
<td>SOBGTR</td>
<td>Subtract one and branch on greater</td>
<td>F5</td>
<td>217</td>
</tr>
<tr>
<td>SPANC</td>
<td>Span characters</td>
<td>2B</td>
<td>242</td>
</tr>
<tr>
<td>SUBB2</td>
<td>Subtract byte 2 operand</td>
<td>82</td>
<td>144</td>
</tr>
<tr>
<td>SUBB3</td>
<td>Subtract byte 3 operand</td>
<td>83</td>
<td>144</td>
</tr>
<tr>
<td>SUBD2</td>
<td>Subtract double 2 operand</td>
<td>62</td>
<td>144</td>
</tr>
<tr>
<td>SUBD3</td>
<td>Subtract double 3 operand</td>
<td>63</td>
<td>144</td>
</tr>
<tr>
<td>SUBF2</td>
<td>Subtract floating 2 operand</td>
<td>42</td>
<td>144</td>
</tr>
<tr>
<td>SUBF3</td>
<td>Subtract floating 3 operand</td>
<td>43</td>
<td>144</td>
</tr>
<tr>
<td>SUBL2</td>
<td>Subtract long 2 operand</td>
<td>C2</td>
<td>144</td>
</tr>
<tr>
<td>SUBL3</td>
<td>Subtract long 3 operand</td>
<td>C3</td>
<td>144</td>
</tr>
<tr>
<td>SUBP4</td>
<td>Subtract packed 4 operand</td>
<td>22</td>
<td>258</td>
</tr>
<tr>
<td>SUBP6</td>
<td>Subtract packed 6 operand</td>
<td>23</td>
<td>258</td>
</tr>
<tr>
<td>SUBW2</td>
<td>Subtract word 2 operand</td>
<td>A2</td>
<td>144</td>
</tr>
<tr>
<td>SUBW3</td>
<td>Subtract word 3 operand</td>
<td>A3</td>
<td>144</td>
</tr>
<tr>
<td>SVPCTX</td>
<td>Save process context</td>
<td>07</td>
<td>—</td>
</tr>
<tr>
<td>TSTB</td>
<td>Test byte</td>
<td>95</td>
<td>139</td>
</tr>
<tr>
<td>TSTD</td>
<td>Test double</td>
<td>73</td>
<td>139</td>
</tr>
<tr>
<td>TSTF</td>
<td>Test float</td>
<td>53</td>
<td>139</td>
</tr>
<tr>
<td>MNEMONIC</td>
<td>INSTRUCTION</td>
<td>OPCODE</td>
<td>PAGE</td>
</tr>
<tr>
<td>----------</td>
<td>---------------------------------</td>
<td>--------</td>
<td>------</td>
</tr>
<tr>
<td>TSTL</td>
<td>Test long</td>
<td>D5</td>
<td>139</td>
</tr>
<tr>
<td>TSTW</td>
<td>Test word</td>
<td>B5</td>
<td>139</td>
</tr>
<tr>
<td>XFC</td>
<td>Extended function call</td>
<td>FC</td>
<td>—</td>
</tr>
<tr>
<td>XORB2</td>
<td>Exclusive OR byte 2 operand</td>
<td>8C</td>
<td>158</td>
</tr>
<tr>
<td>XORB3</td>
<td>Exclusive OR byte 3 operand</td>
<td>8D</td>
<td>158</td>
</tr>
<tr>
<td>XORL2</td>
<td>Exclusive OR long 2 operand</td>
<td>CC</td>
<td>158</td>
</tr>
<tr>
<td>XORL3</td>
<td>Exclusive OR long 3 operand</td>
<td>CD</td>
<td>158</td>
</tr>
<tr>
<td>XORW2</td>
<td>Exclusive OR word 2 operand</td>
<td>TC</td>
<td>158</td>
</tr>
<tr>
<td>XORW3</td>
<td>Exclusive OR word 3 operand</td>
<td>AD</td>
<td>158</td>
</tr>
<tr>
<td></td>
<td><em>Reserved to DIGITAL</em></td>
<td>57</td>
<td></td>
</tr>
<tr>
<td></td>
<td><em>Reserved to DIGITAL</em></td>
<td>59</td>
<td></td>
</tr>
<tr>
<td></td>
<td><em>Reserved to DIGITAL</em></td>
<td>5A</td>
<td></td>
</tr>
<tr>
<td></td>
<td><em>Reserved to DIGITAL</em></td>
<td>5B</td>
<td></td>
</tr>
<tr>
<td></td>
<td><em>Reserved to DIGITAL</em></td>
<td>77</td>
<td></td>
</tr>
<tr>
<td>ESCD</td>
<td><em>Reserved to DIGITAL</em></td>
<td>FD</td>
<td></td>
</tr>
<tr>
<td>ESCE</td>
<td><em>Reserved to DIGITAL</em></td>
<td>FE</td>
<td></td>
</tr>
<tr>
<td>ESCF</td>
<td><em>Reserved to DIGITAL</em></td>
<td>FF</td>
<td></td>
</tr>
<tr>
<td>OPCODE</td>
<td>MNEMONIC</td>
<td>INSTRUCTION</td>
<td></td>
</tr>
<tr>
<td>--------</td>
<td>-------------</td>
<td>------------------------------------------------</td>
<td></td>
</tr>
<tr>
<td>00</td>
<td>HALT</td>
<td>Halt</td>
<td></td>
</tr>
<tr>
<td>01</td>
<td>NOP</td>
<td>No operation</td>
<td></td>
</tr>
<tr>
<td>02</td>
<td>REI</td>
<td>Return from exception or interrupt</td>
<td></td>
</tr>
<tr>
<td>03</td>
<td>BPT</td>
<td>Break point fault</td>
<td></td>
</tr>
<tr>
<td>04</td>
<td>RET</td>
<td>Return from called procedure</td>
<td></td>
</tr>
<tr>
<td>05</td>
<td>RSB</td>
<td>Return from subroutine</td>
<td></td>
</tr>
<tr>
<td>06</td>
<td>LDPCTX</td>
<td>Load process context</td>
<td></td>
</tr>
<tr>
<td>07</td>
<td>SVPCTX</td>
<td>Save process context</td>
<td></td>
</tr>
<tr>
<td>08</td>
<td>CVTPS</td>
<td>Convert packed to leading separate numeric</td>
<td></td>
</tr>
<tr>
<td>09</td>
<td>CVTSP</td>
<td>Convert leading separate numeric to packed</td>
<td></td>
</tr>
<tr>
<td>OA</td>
<td>INDEX</td>
<td>Compute index</td>
<td></td>
</tr>
<tr>
<td>OB</td>
<td>CRC</td>
<td>Calculate cyclic redundancy check</td>
<td></td>
</tr>
<tr>
<td>OC</td>
<td>PROBER</td>
<td>Probe read access</td>
<td></td>
</tr>
<tr>
<td>OD</td>
<td>PROBEW</td>
<td>Prove write access</td>
<td></td>
</tr>
<tr>
<td>OE</td>
<td>INSQUE</td>
<td>Insert into queue</td>
<td></td>
</tr>
<tr>
<td>OF</td>
<td>REMQUE</td>
<td>Remove from queue</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>BSBB</td>
<td>Branch to subroutine with byte displacement</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>BRB</td>
<td>Branch with byte displacement</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>BNEQ, BNEQU</td>
<td>Branch on not equal,</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Branch on not equal unsigned</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>BEQL, BEQLU</td>
<td>Branch on equal, Branch on equal</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>unsigned</td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>BGTR</td>
<td>Branch on greater</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>BLEQ</td>
<td>Branch on less or equal</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>JSB</td>
<td>Jump to subroutine</td>
<td></td>
</tr>
<tr>
<td>17</td>
<td>JMP</td>
<td>Jump</td>
<td></td>
</tr>
<tr>
<td>18</td>
<td>BGEQ</td>
<td>Branch on greater or equal</td>
<td></td>
</tr>
<tr>
<td>19</td>
<td>BLSS</td>
<td>Branch on less</td>
<td></td>
</tr>
<tr>
<td>1A</td>
<td>BGTRU</td>
<td>Branch on greater unsigned</td>
<td></td>
</tr>
<tr>
<td>1B</td>
<td>BLEQU</td>
<td>Branch on less or equal unsigned</td>
<td></td>
</tr>
<tr>
<td>1C</td>
<td>BVC</td>
<td>Branch on overflow clear</td>
<td></td>
</tr>
<tr>
<td>1D</td>
<td>BVS</td>
<td>Branch on overflow set</td>
<td></td>
</tr>
<tr>
<td>1E</td>
<td>BGEQU, BCC</td>
<td>Branch on greater or equal unsigned,</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Branch on carry clear</td>
<td></td>
</tr>
<tr>
<td>OPCODE</td>
<td>MNEMONIC</td>
<td>INSTRUCTION</td>
<td></td>
</tr>
<tr>
<td>--------</td>
<td>------------</td>
<td>-----------------------------------------------------------</td>
<td></td>
</tr>
<tr>
<td>1F</td>
<td>BLSSU, BCS</td>
<td>Branch on less unsigned, Branch on carry set</td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>ADDP4</td>
<td>Add packed 4 operand</td>
<td></td>
</tr>
<tr>
<td>21</td>
<td>ADDP6</td>
<td>Add packed 6 operand</td>
<td></td>
</tr>
<tr>
<td>22</td>
<td>SUBP4</td>
<td>Subtract packed 4 operand</td>
<td></td>
</tr>
<tr>
<td>23</td>
<td>SUBP6</td>
<td>Subtract packed 6 operand</td>
<td></td>
</tr>
<tr>
<td>24</td>
<td>CVTPT</td>
<td>Convert packed to trailing numeric</td>
<td></td>
</tr>
<tr>
<td>25</td>
<td>MULP</td>
<td>Multiply packed</td>
<td></td>
</tr>
<tr>
<td>26</td>
<td>CVTTP</td>
<td>Convert trailing numeric to packed</td>
<td></td>
</tr>
<tr>
<td>27</td>
<td>DIVP</td>
<td>Divide packed</td>
<td></td>
</tr>
<tr>
<td>28</td>
<td>MOVNC3</td>
<td>Move character 3 operand</td>
<td></td>
</tr>
<tr>
<td>29</td>
<td>CMPC3</td>
<td>Compare character 3 operand</td>
<td></td>
</tr>
<tr>
<td>2A</td>
<td>SCANCE</td>
<td>Scan for character</td>
<td></td>
</tr>
<tr>
<td>2B</td>
<td>SPANCE</td>
<td>Span characters</td>
<td></td>
</tr>
<tr>
<td>2C</td>
<td>MOVNC5</td>
<td>Move character 5 operand</td>
<td></td>
</tr>
<tr>
<td>2D</td>
<td>CMPC5</td>
<td>Compare character 5 operand</td>
<td></td>
</tr>
<tr>
<td>2E</td>
<td>MOVTC</td>
<td>Move translated characters</td>
<td></td>
</tr>
<tr>
<td>2F</td>
<td>MOVTUC</td>
<td>Move translated until character</td>
<td></td>
</tr>
<tr>
<td>30</td>
<td>BSBW</td>
<td>Branch to subroutine with word displacement</td>
<td></td>
</tr>
<tr>
<td>31</td>
<td>BRW</td>
<td>Branch with word displacement</td>
<td></td>
</tr>
<tr>
<td>32</td>
<td>CVTWL</td>
<td>Convert word to long</td>
<td></td>
</tr>
<tr>
<td>33</td>
<td>CVTWB</td>
<td>Convert word to byte</td>
<td></td>
</tr>
<tr>
<td>34</td>
<td>MOV</td>
<td>Move packed</td>
<td></td>
</tr>
<tr>
<td>35</td>
<td>CMPP3</td>
<td>Compare packed 3 operand</td>
<td></td>
</tr>
<tr>
<td>36</td>
<td>CVTPL</td>
<td>Convert packed to long</td>
<td></td>
</tr>
<tr>
<td>37</td>
<td>CMPP4</td>
<td>Compare packed 4 operand</td>
<td></td>
</tr>
<tr>
<td>38</td>
<td>EDITPC</td>
<td>Edit packed to character</td>
<td></td>
</tr>
<tr>
<td>39</td>
<td>MATCHC</td>
<td>Match characters</td>
<td></td>
</tr>
<tr>
<td>3A</td>
<td>LOCC</td>
<td>Locate character</td>
<td></td>
</tr>
<tr>
<td>3B</td>
<td>SKPC</td>
<td>Skip character</td>
<td></td>
</tr>
<tr>
<td>3C</td>
<td>MOVZWL</td>
<td>Move zero-extended word to long</td>
<td></td>
</tr>
<tr>
<td>3D</td>
<td>ACBW</td>
<td>Add compare and branch word</td>
<td></td>
</tr>
<tr>
<td>3E</td>
<td>MOVAW</td>
<td>Move address of word</td>
<td></td>
</tr>
<tr>
<td>3F</td>
<td>PUSHAW</td>
<td>Push address of word</td>
<td></td>
</tr>
<tr>
<td>40</td>
<td>ADDF2</td>
<td>Add floating 2 operand</td>
<td></td>
</tr>
<tr>
<td>41</td>
<td>ADDF3</td>
<td>Add floating 3 operand</td>
<td></td>
</tr>
<tr>
<td>OPCODE</td>
<td>MNEMONIC</td>
<td>INSTRUCTION</td>
<td></td>
</tr>
<tr>
<td>--------</td>
<td>----------</td>
<td>--------------------------------------------------</td>
<td></td>
</tr>
<tr>
<td>42</td>
<td>SUBF2</td>
<td>Subtract floating 2 operand</td>
<td></td>
</tr>
<tr>
<td>43</td>
<td>SUBF3</td>
<td>Subtract floating 3 operand</td>
<td></td>
</tr>
<tr>
<td>44</td>
<td>MULF2</td>
<td>Multiply floating 2 operand</td>
<td></td>
</tr>
<tr>
<td>45</td>
<td>MULF3</td>
<td>Multiply floating 3 operand</td>
<td></td>
</tr>
<tr>
<td>46</td>
<td>DIVF2</td>
<td>Divide floating 2 operand</td>
<td></td>
</tr>
<tr>
<td>47</td>
<td>DIVF3</td>
<td>Divide floating 3 operand</td>
<td></td>
</tr>
<tr>
<td>48</td>
<td>CVTFB</td>
<td>Convert float to byte</td>
<td></td>
</tr>
<tr>
<td>49</td>
<td>CVTFW</td>
<td>Convert float to word</td>
<td></td>
</tr>
<tr>
<td>4A</td>
<td>CVTFL</td>
<td>Convert float to long</td>
<td></td>
</tr>
<tr>
<td>4B</td>
<td>CVTRFL</td>
<td>Convert rounded float to long</td>
<td></td>
</tr>
<tr>
<td>4C</td>
<td>CVTBF</td>
<td>Convert byte to float</td>
<td></td>
</tr>
<tr>
<td>4D</td>
<td>CVTWF</td>
<td>Convert word to float</td>
<td></td>
</tr>
<tr>
<td>4E</td>
<td>CVTLF</td>
<td>Convert long to float</td>
<td></td>
</tr>
<tr>
<td>4F</td>
<td>ACBF</td>
<td>Add compare and branch floating</td>
<td></td>
</tr>
<tr>
<td>50</td>
<td>MOVF</td>
<td>Move float</td>
<td></td>
</tr>
<tr>
<td>51</td>
<td>CMPF</td>
<td>Compare floating</td>
<td></td>
</tr>
<tr>
<td>52</td>
<td>MNEGFI</td>
<td>Move negated floating</td>
<td></td>
</tr>
<tr>
<td>53</td>
<td>TSTF</td>
<td>Test float</td>
<td></td>
</tr>
<tr>
<td>54</td>
<td>EMODF</td>
<td>Extended modulus floating</td>
<td></td>
</tr>
<tr>
<td>55</td>
<td>POLYF</td>
<td>Evaluate polynomial floating</td>
<td></td>
</tr>
<tr>
<td>56</td>
<td>CVTFD</td>
<td>Convert float to double</td>
<td></td>
</tr>
<tr>
<td>57</td>
<td></td>
<td>RESERVED to DIGITAL</td>
<td></td>
</tr>
<tr>
<td>58</td>
<td>ADAWI</td>
<td>Add aligned word interlocked</td>
<td></td>
</tr>
<tr>
<td>59</td>
<td></td>
<td>RESERVED to DIGITAL</td>
<td></td>
</tr>
<tr>
<td>5A</td>
<td></td>
<td>RESERVED to DIGITAL</td>
<td></td>
</tr>
<tr>
<td>5B</td>
<td></td>
<td>RESERVED to DIGITAL</td>
<td></td>
</tr>
<tr>
<td>5C</td>
<td>INSQHI</td>
<td>Insert into queue head, interlocked</td>
<td></td>
</tr>
<tr>
<td>5D</td>
<td>INSQTI</td>
<td>Insert into queue tail, interlocked</td>
<td></td>
</tr>
<tr>
<td>5E</td>
<td>REMQHI</td>
<td>Remove from queue head, interlocked</td>
<td></td>
</tr>
<tr>
<td>5F</td>
<td>REMQTI</td>
<td>Remove from queue tail, interlocked</td>
<td></td>
</tr>
<tr>
<td>60</td>
<td>ADDD2</td>
<td>Add double 2 operand</td>
<td></td>
</tr>
<tr>
<td>61</td>
<td>ADDD3</td>
<td>Add double 3 operand</td>
<td></td>
</tr>
<tr>
<td>62</td>
<td>SUBD2</td>
<td>Subtract double 2 operand</td>
<td></td>
</tr>
<tr>
<td>63</td>
<td>SUBD3</td>
<td>Subtract double 3 operand</td>
<td></td>
</tr>
<tr>
<td>64</td>
<td>MULD2</td>
<td>Multiply double 2 operand</td>
<td></td>
</tr>
<tr>
<td>65</td>
<td>MULD3</td>
<td>Multiply double 3 operand</td>
<td></td>
</tr>
<tr>
<td>66</td>
<td>DIVD2</td>
<td>Divide double 2 operand</td>
<td></td>
</tr>
<tr>
<td>67</td>
<td>DIVD3</td>
<td>Divide double 3 operand</td>
<td></td>
</tr>
</tbody>
</table>

343
<table>
<thead>
<tr>
<th>OPCODE</th>
<th>MNEMONIC</th>
<th>INSTRUCTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>68</td>
<td>CVTDB</td>
<td>Convert double to byte</td>
</tr>
<tr>
<td>69</td>
<td>CVTDW</td>
<td>Convert double to word</td>
</tr>
<tr>
<td>6A</td>
<td>CVTDL</td>
<td>Convert double to long</td>
</tr>
<tr>
<td>6B</td>
<td>CVTRDL</td>
<td>Convert rounded double to long</td>
</tr>
<tr>
<td>6C</td>
<td>CVTBD</td>
<td>Convert byte to double</td>
</tr>
<tr>
<td>6D</td>
<td>CVTWD</td>
<td>Convert word to double</td>
</tr>
<tr>
<td>6E</td>
<td>CVTLD</td>
<td>Convert long to double</td>
</tr>
<tr>
<td>6F</td>
<td>ACBD</td>
<td>Add compare and branch double</td>
</tr>
<tr>
<td>70</td>
<td>MOVD</td>
<td>Move double</td>
</tr>
<tr>
<td>71</td>
<td>CMPD</td>
<td>Compare double</td>
</tr>
<tr>
<td>72</td>
<td>MNEGDP</td>
<td>Move negated double</td>
</tr>
<tr>
<td>73</td>
<td>TSTD</td>
<td>Test double</td>
</tr>
<tr>
<td>74</td>
<td>EMOODD</td>
<td>Extended modulus double</td>
</tr>
<tr>
<td>75</td>
<td>POLYD</td>
<td>Evaluate polynomial double</td>
</tr>
<tr>
<td>76</td>
<td>CVTDF</td>
<td>Convert double to float</td>
</tr>
<tr>
<td>77</td>
<td></td>
<td>RESERVED to DIGITAL</td>
</tr>
<tr>
<td>78</td>
<td>ASHL</td>
<td>Arithmetic shift long</td>
</tr>
<tr>
<td>79</td>
<td>ASHQ</td>
<td>Arithmetic shift quad</td>
</tr>
<tr>
<td>7A</td>
<td>EMUL</td>
<td>Extended multiply</td>
</tr>
<tr>
<td>7B</td>
<td>EDIV</td>
<td>Extended divide</td>
</tr>
<tr>
<td>7C</td>
<td>CLRQ, CLRD</td>
<td>Clear quad, Clear double</td>
</tr>
<tr>
<td>7D</td>
<td>MOVQ</td>
<td>Move quad</td>
</tr>
<tr>
<td>7E</td>
<td>MOVAQ, MOVAD</td>
<td>Move address of quad, Move address of double</td>
</tr>
<tr>
<td>7F</td>
<td>PUSAQ, PUSHAD</td>
<td>Push address of quad, Push address of double</td>
</tr>
<tr>
<td>80</td>
<td>ADDB2</td>
<td>Add byte 2 operand</td>
</tr>
<tr>
<td>81</td>
<td>ADDB3</td>
<td>Add byte 3 operand</td>
</tr>
<tr>
<td>82</td>
<td>SUBB2</td>
<td>Subtract byte 2 operand</td>
</tr>
<tr>
<td>83</td>
<td>SUBB3</td>
<td>Subtract byte 3 operand</td>
</tr>
<tr>
<td>84</td>
<td>MULB2</td>
<td>Multiply byte 2 operand</td>
</tr>
<tr>
<td>85</td>
<td>MULB3</td>
<td>Multiply byte 3 operand</td>
</tr>
<tr>
<td>86</td>
<td>DIVB2</td>
<td>Divide byte 2 operand</td>
</tr>
<tr>
<td>87</td>
<td>DIVB3</td>
<td>Divide byte 3 operand</td>
</tr>
<tr>
<td>88</td>
<td>BISB2</td>
<td>Bit set byte 2 operand</td>
</tr>
<tr>
<td>89</td>
<td>BISB3</td>
<td>Bit set byte 3 operand</td>
</tr>
<tr>
<td>8A</td>
<td>BICB2</td>
<td>Bit clear byte 2 operand</td>
</tr>
<tr>
<td>8B</td>
<td>BICB3</td>
<td>Bit clear byte 3 operand</td>
</tr>
<tr>
<td>8C</td>
<td>XORB2</td>
<td>Exclusive OR byte 2 operand</td>
</tr>
</tbody>
</table>

344
<table>
<thead>
<tr>
<th>OPCODE</th>
<th>MNEMONIC</th>
<th>INSTRUCTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>8D</td>
<td>XORB3</td>
<td>Exclusive OR byte 3 operand</td>
</tr>
<tr>
<td>8E</td>
<td>MNEGB</td>
<td>Move negated byte</td>
</tr>
<tr>
<td>8F</td>
<td>CASEB</td>
<td>Case byte</td>
</tr>
<tr>
<td>90</td>
<td>MOVB</td>
<td>Move byte</td>
</tr>
<tr>
<td>91</td>
<td>CMPB</td>
<td>Compare byte</td>
</tr>
<tr>
<td>92</td>
<td>MCOMB</td>
<td>Move complemented byte</td>
</tr>
<tr>
<td>93</td>
<td>BITB</td>
<td>Bit test byte</td>
</tr>
<tr>
<td>94</td>
<td>CLR B</td>
<td>Clear byte</td>
</tr>
<tr>
<td>95</td>
<td>TST B</td>
<td>Test byte</td>
</tr>
<tr>
<td>96</td>
<td>INCB</td>
<td>Increment byte</td>
</tr>
<tr>
<td>97</td>
<td>DECB</td>
<td>Decrement byte</td>
</tr>
<tr>
<td>98</td>
<td>CVTBL</td>
<td>Convert byte long</td>
</tr>
<tr>
<td>99</td>
<td>CVTBW</td>
<td>Convert byte to word</td>
</tr>
<tr>
<td>9A</td>
<td>MOVZBL</td>
<td>Move zero-extended byte to long</td>
</tr>
<tr>
<td>9B</td>
<td>MOVZBW</td>
<td>Move zero-extended byte to word</td>
</tr>
<tr>
<td>9C</td>
<td>ROTL</td>
<td>Rotate long</td>
</tr>
<tr>
<td>9D</td>
<td>ACBB</td>
<td>Add compare and branch byte</td>
</tr>
<tr>
<td>9E</td>
<td>MOVAB</td>
<td>Move address of byte</td>
</tr>
<tr>
<td>9F</td>
<td>PUSHAB</td>
<td>Push address of byte</td>
</tr>
<tr>
<td>A0</td>
<td>ADDW2</td>
<td>Add word 2 operand</td>
</tr>
<tr>
<td>A1</td>
<td>ADDW3</td>
<td>Add word 3 operand</td>
</tr>
<tr>
<td>A2</td>
<td>SUBW2</td>
<td>Subtract word 2 operand</td>
</tr>
<tr>
<td>A3</td>
<td>SUBW3</td>
<td>Subtract word 3 operand</td>
</tr>
<tr>
<td>A4</td>
<td>MULW2</td>
<td>Multiply word 2 operand</td>
</tr>
<tr>
<td>A5</td>
<td>MULW3</td>
<td>Multiply word 3 operand</td>
</tr>
<tr>
<td>A6</td>
<td>DIVW2</td>
<td>Divide word 2 operand</td>
</tr>
<tr>
<td>A7</td>
<td>DIVW3</td>
<td>Divide word 3 operand</td>
</tr>
<tr>
<td>A8</td>
<td>BISW2</td>
<td>Bit set word 2 operand</td>
</tr>
<tr>
<td>A9</td>
<td>BISW3</td>
<td>Bit set word 3 operand</td>
</tr>
<tr>
<td>AA</td>
<td>BICW2</td>
<td>Bit clear word 2 operand</td>
</tr>
<tr>
<td>AB</td>
<td>BICW3</td>
<td>Bit clear word 3 operand</td>
</tr>
<tr>
<td>AC</td>
<td>XORW2</td>
<td>Exclusive OR word 2 operand</td>
</tr>
<tr>
<td>AD</td>
<td>XORW3</td>
<td>Exclusive OR word 3 operand</td>
</tr>
<tr>
<td>AE</td>
<td>MNEGW</td>
<td>Move negated word</td>
</tr>
<tr>
<td>AF</td>
<td>CASEW</td>
<td>Case word</td>
</tr>
<tr>
<td>B0</td>
<td>MOVW</td>
<td>Move word</td>
</tr>
<tr>
<td>B1</td>
<td>CMPW</td>
<td>Compare word</td>
</tr>
<tr>
<td>OPCODE</td>
<td>MNEMONIC</td>
<td>INSTRUCTION</td>
</tr>
<tr>
<td>--------</td>
<td>------------</td>
<td>----------------------------------</td>
</tr>
<tr>
<td>B2</td>
<td>MCOMW</td>
<td>Move complemented word</td>
</tr>
<tr>
<td>B3</td>
<td>BITW</td>
<td>Bit test word</td>
</tr>
<tr>
<td>B4</td>
<td>CLRW</td>
<td>Clear word</td>
</tr>
<tr>
<td>B5</td>
<td>TSTW</td>
<td>Test word</td>
</tr>
<tr>
<td>B6</td>
<td>INCW</td>
<td>Increment word</td>
</tr>
<tr>
<td>B7</td>
<td>DECW</td>
<td>Decrement word</td>
</tr>
<tr>
<td>B8</td>
<td>BISPSW</td>
<td>Bit set processor status word</td>
</tr>
<tr>
<td>B9</td>
<td>BICPSW</td>
<td>Bit clear processor status word</td>
</tr>
<tr>
<td>BA</td>
<td>POPR</td>
<td>Pop register</td>
</tr>
<tr>
<td>BB</td>
<td>PUSHR</td>
<td>Push register</td>
</tr>
<tr>
<td>BC</td>
<td>CHMK</td>
<td>Change mode to kernel</td>
</tr>
<tr>
<td>BD</td>
<td>CHME</td>
<td>Change mode to executive</td>
</tr>
<tr>
<td>BE</td>
<td>CHMS</td>
<td>Change mode to supervisor</td>
</tr>
<tr>
<td>BF</td>
<td>CHMU</td>
<td>Change mode to user</td>
</tr>
<tr>
<td>C0</td>
<td>ADDL2</td>
<td>Add long 2 operand</td>
</tr>
<tr>
<td>C1</td>
<td>ADDL3</td>
<td>Add long 3 operand</td>
</tr>
<tr>
<td>C2</td>
<td>SUBL2</td>
<td>Subtract long 2 operand</td>
</tr>
<tr>
<td>C3</td>
<td>SUBL3</td>
<td>Subtract long 3 operand</td>
</tr>
<tr>
<td>C4</td>
<td>MULL2</td>
<td>Multiply long 2 operand</td>
</tr>
<tr>
<td>C5</td>
<td>MULL3</td>
<td>Multiply long 3 operand</td>
</tr>
<tr>
<td>C6</td>
<td>DIVL2</td>
<td>Divide long 2 operand</td>
</tr>
<tr>
<td>C7</td>
<td>DIVL3</td>
<td>Divide long 3 operand</td>
</tr>
<tr>
<td>C8</td>
<td>BISL2</td>
<td>Bit set long 2 operand</td>
</tr>
<tr>
<td>C9</td>
<td>BISL3</td>
<td>Bit set long 3 operand</td>
</tr>
<tr>
<td>CA</td>
<td>BICL2</td>
<td>Bit clear long 2 operand</td>
</tr>
<tr>
<td>CB</td>
<td>BICL3</td>
<td>Bit clear long 3 operand</td>
</tr>
<tr>
<td>CC</td>
<td>XORL2</td>
<td>Exclusive OR long 2 operand</td>
</tr>
<tr>
<td>CD</td>
<td>XORL3</td>
<td>Exclusive OR long 3 operand</td>
</tr>
<tr>
<td>CE</td>
<td>MNEGL</td>
<td>Move negated long</td>
</tr>
<tr>
<td>CF</td>
<td>CASEL</td>
<td>Case long</td>
</tr>
<tr>
<td>D0</td>
<td>MOVL</td>
<td>Move long</td>
</tr>
<tr>
<td>D1</td>
<td>CMPL</td>
<td>Compare long</td>
</tr>
<tr>
<td>D2</td>
<td>MCOMML</td>
<td>Move complemented long</td>
</tr>
<tr>
<td>D3</td>
<td>BITL</td>
<td>Bit test long</td>
</tr>
<tr>
<td>D4</td>
<td>CLRL, CLRF</td>
<td>Clear long, Clear float</td>
</tr>
<tr>
<td>D5</td>
<td>TSTL</td>
<td>Test long</td>
</tr>
<tr>
<td>D6</td>
<td>INCL</td>
<td>Increment long</td>
</tr>
<tr>
<td>D7</td>
<td>DECL</td>
<td>Decrement long</td>
</tr>
</tbody>
</table>

346
<table>
<thead>
<tr>
<th>OPCODE</th>
<th>MNEMONIC</th>
<th>INSTRUCTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>D8</td>
<td>ADWC</td>
<td>Add with carry</td>
</tr>
<tr>
<td>D9</td>
<td>SBWC</td>
<td>Subtract with carry</td>
</tr>
<tr>
<td>DA</td>
<td>MTPR</td>
<td>Move to processor register</td>
</tr>
<tr>
<td>DB</td>
<td>MFPR</td>
<td>Move from processor register</td>
</tr>
<tr>
<td>DC</td>
<td>MOVPSL</td>
<td>Move processor status longword</td>
</tr>
<tr>
<td>DD</td>
<td>PUSHL</td>
<td>Push long</td>
</tr>
<tr>
<td>DE</td>
<td>MOVAL, MOVAF</td>
<td>Move address of long, Move address of float</td>
</tr>
<tr>
<td>DF</td>
<td>PUSHL&lt; PUSHAF</td>
<td>Push address of long, Push address of float</td>
</tr>
<tr>
<td>E0</td>
<td>BBS</td>
<td>Branch on bit set</td>
</tr>
<tr>
<td>E1</td>
<td>BBC</td>
<td>Branch on bit clear</td>
</tr>
<tr>
<td>E2</td>
<td>BBSS</td>
<td>Branch on bit set and set</td>
</tr>
<tr>
<td>E3</td>
<td>BBCS</td>
<td>Branch on bit clear and set</td>
</tr>
<tr>
<td>E4</td>
<td>BBSC</td>
<td>Branch on bit set and clear</td>
</tr>
<tr>
<td>E5</td>
<td>BBCC</td>
<td>Branch on bit clear and clear</td>
</tr>
<tr>
<td>E6</td>
<td>BBSSI</td>
<td>Branch on bit set and set interlocked</td>
</tr>
<tr>
<td>E7</td>
<td>BBCCI</td>
<td>Branch on bit clear and clear interlocked</td>
</tr>
<tr>
<td>IE8</td>
<td>BLBS</td>
<td>Branch on low bit set</td>
</tr>
<tr>
<td>E9</td>
<td>BLBC</td>
<td>Branch on low bit clear</td>
</tr>
<tr>
<td>EA</td>
<td>FFS</td>
<td>Find first set bit</td>
</tr>
<tr>
<td>EB</td>
<td>FFC</td>
<td>Find first clear bit</td>
</tr>
<tr>
<td>EC</td>
<td>CMPV</td>
<td>Compare field</td>
</tr>
<tr>
<td>ED</td>
<td>CMPZV</td>
<td>Compare zero-extended field</td>
</tr>
<tr>
<td>EE</td>
<td>EXTV</td>
<td>Extract field</td>
</tr>
<tr>
<td>EF</td>
<td>EXTZV</td>
<td>Extract zero-extended field</td>
</tr>
<tr>
<td>F0</td>
<td>INSV</td>
<td>Insert field</td>
</tr>
<tr>
<td>F1</td>
<td>ACBL</td>
<td>Add compare and branch long</td>
</tr>
<tr>
<td>F2</td>
<td>AOBLLSS</td>
<td>Add one and branch on less</td>
</tr>
<tr>
<td>F3</td>
<td>AOBLEQ</td>
<td>Add one and branch on less or equal</td>
</tr>
<tr>
<td>F4</td>
<td>SOBGEQ</td>
<td>Subtract one and branch on greater or equal</td>
</tr>
<tr>
<td>F5</td>
<td>SOBGTR</td>
<td>Subtract one and branch on greater</td>
</tr>
<tr>
<td>F6</td>
<td>CVTLB</td>
<td>Convert long to byte</td>
</tr>
<tr>
<td>F7</td>
<td>CVTLW</td>
<td>Convert long to word</td>
</tr>
<tr>
<td>F8</td>
<td>ASHP</td>
<td>Arithmetic shift and round packed</td>
</tr>
<tr>
<td>F9</td>
<td>CVTLP</td>
<td>Convert long to packed</td>
</tr>
<tr>
<td>FA</td>
<td>CALLG</td>
<td>Call with general argument list</td>
</tr>
</tbody>
</table>
### Appendix B

<table>
<thead>
<tr>
<th>OPCODE</th>
<th>MNEMONIC</th>
<th>INSTRUCTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>FB</td>
<td>CALLS</td>
<td>Call with stack</td>
</tr>
<tr>
<td>FC</td>
<td>XFC</td>
<td>Extended function call</td>
</tr>
<tr>
<td>FD</td>
<td>ESCD to DIGITAL</td>
<td></td>
</tr>
<tr>
<td>FE</td>
<td>ESCE to DIGITAL</td>
<td></td>
</tr>
<tr>
<td>FF</td>
<td>ESCF to DIGITAL</td>
<td></td>
</tr>
</tbody>
</table>
APPENDIX C
PROCEDURE CALLING AND CONDITION HANDLING

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 for the use of the function value as a condition value to indicate success or failure.
• register usage—which registers are preserved and who is responsible for preserving them.
• stack usage—rules governing the use of the stack.
• data types of arguments—the types of all arguments that can be passed.
• argument (or parameter) descriptors—how descriptors are passed for the more complex arguments.
• condition handling—how exceptional conditions are signaled and how they can be handled in a modular fashion.
• stack unwinding—how the current thread of execution can be aborted cleanly.

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
Appendix C

standard considers the requirements of BASIC, COBOL, VAX-11 FORTRAN, FORTRAN IV-PLUS, BLISS, VAX-11 MACRO, 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, VAX-11 MACRO, operating system) that cannot be invoked from the higher-level languages.

2. The calling program and called procedure can be written in different languages.

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 (1) 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.

CALLING SEQUENCE

At the option of the calling program, the called procedure may be invoked using either the CALLG or CALLS instruction:

CALLG arglist, proc
CALLS argcnt, proc

CALLS pushes the argument count Argcnt onto the stack as a long-word and sets the argument pointer (AP) to the top of the stack. The complete sequence using CALLS is thus:

    push argn
    ....
    push arg1
    CALLS #n, proc
Appendix C

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 stack unwind operations.

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

**ARGUMENT LIST**
The argument list in the primary means of passing information to and receiving results from a procedure.

**Argument List Format**
The format of the argument list is a sequence of longwords:

```
<table>
<thead>
<tr>
<th>0</th>
<th>n</th>
</tr>
</thead>
<tbody>
<tr>
<td>ARG 1</td>
<td></td>
</tr>
<tr>
<td>ARG 2</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>ARG n</td>
<td></td>
</tr>
</tbody>
</table>
```

The argument count n is an unsigned byte in the first byte of the argument 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 parameters to be passed by-value, by-reference, 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 CALLe d with a list consisting of a 0 argument count longword. This is accomplished as follows:

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

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

**Argument Lists And Higher-level Languages**
Higher-level language functional notations for procedure CALLs are mapped into VAX-11 argument list according to the following rules:

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

**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, these language compilers can 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.

**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 passes scalars, arrays, and functions by-reference, and passes strings (CHARACTER) 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.
Appendix C

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, arg, 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 necessary 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.

FUNCTION VALUE RETURN
A function value is returned in register R0 if representable in 32-bits, and register R0 and R1 if representable in 64 bits. Two separate 32-bit entities cannot be returned in R0 and R1 because higher-level languages cannot process them. 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 (i.e., not on the 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 as described in the next section.
Appendix C

CONDITION VALUE
VAX-11 uses a standard means to report the success or failure of a called procedure and to describe an exception condition 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:

```
<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>28 27</td>
<td>3 2 0</td>
</tr>
<tr>
<td>CNTRL</td>
<td>CONDITION IDENTIFICATION</td>
<td>SEV</td>
</tr>
</tbody>
</table>
```

```
<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>27</td>
<td>16 15</td>
<td>3</td>
</tr>
<tr>
<td>FACILITY NUMBER</td>
<td>MESSAGE NUMBER</td>
<td></td>
</tr>
</tbody>
</table>
```

**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**
A status identification; i.e., is a description of the hardware exception that occurred or a software defined value. Message numbers with bit 15 set are specific to a single facility. Message numbers with bit 15 clear are system wide status codes.

**severity**
The severity code bit 0 is set for success (logical true) and is clear for failure (logical false); bits 1 and 2 distinguishes degrees of success or failure. Thus, the three bits, 0 through 2, can be considered as follows:
Appendix C

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

cntrl

Four control bits:
STS$V_INHIB_MSG (list name)

Bit 28 inhibits the message associated with the condition value from being printed by the $EXIT system service. This bit is set by the system default handler after it has output an error message using SYS$PUTMSG system service. It should also be set in the condition value returned by a procedure as a function value, if the procedure has also signaled the condition (so that the condition has been either printed or suppressed).

Bits 29 through 31 must be zero; they are reserved for future use by DIGITAL.

Software symbols are defined for these fields as follows:

<table>
<thead>
<tr>
<th>MNEMONIC</th>
<th>VALUE</th>
<th>MEANING</th>
<th>FIELD</th>
</tr>
</thead>
<tbody>
<tr>
<td>STS$V_COND_ID</td>
<td>3</td>
<td>position of 27:3</td>
<td></td>
</tr>
<tr>
<td>STS$S_COND_ID</td>
<td>25</td>
<td>size of 27:3</td>
<td>condition</td>
</tr>
<tr>
<td>STS$M_COND_ID</td>
<td>mask</td>
<td>mask for 27:3</td>
<td>identification</td>
</tr>
<tr>
<td>STS$V_INHIB_MSG</td>
<td>1@28</td>
<td>position for 28</td>
<td></td>
</tr>
<tr>
<td>STS$S_INHIB_MSG</td>
<td>1</td>
<td>size for 28</td>
<td>inhibit message</td>
</tr>
<tr>
<td>STS$M_INHIB_MSG</td>
<td>mask</td>
<td>mask for 28</td>
<td>on image</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>STS$V_FAC_NO</td>
<td>16</td>
<td>position of 27:16</td>
<td></td>
</tr>
<tr>
<td>STS$S_FAC_NO</td>
<td>12</td>
<td>size of 27:16</td>
<td>facility number</td>
</tr>
<tr>
<td>STS$M_FAC_NO</td>
<td>mask</td>
<td>mask for 27:16</td>
<td></td>
</tr>
<tr>
<td>STS$V_CUST_DEF</td>
<td>27</td>
<td>position for 27</td>
<td>customer facility</td>
</tr>
<tr>
<td>STS$S_CUST_DEF</td>
<td>1</td>
<td>size for 27</td>
<td></td>
</tr>
<tr>
<td>STS$M_CUST_DEF</td>
<td>1@27</td>
<td>mask for 27</td>
<td></td>
</tr>
<tr>
<td>STS$V_MSG_NO</td>
<td>3</td>
<td>position of 15:3</td>
<td></td>
</tr>
<tr>
<td>STS$S_MSG_NO</td>
<td>13</td>
<td>size of 15:3</td>
<td>message number</td>
</tr>
<tr>
<td>STS$M_MSG_NO</td>
<td>mask</td>
<td>mask for 15:3</td>
<td></td>
</tr>
<tr>
<td>STS$V_FAC_SP</td>
<td>15</td>
<td>position of 15</td>
<td></td>
</tr>
<tr>
<td>STS$S_FAC_SP</td>
<td>1</td>
<td>size of 15</td>
<td>facility specific</td>
</tr>
<tr>
<td>STS$M_FAC_SP</td>
<td>1@15</td>
<td>mask for 15</td>
<td></td>
</tr>
</tbody>
</table>
Appendix C

<table>
<thead>
<tr>
<th>MNEMONIC</th>
<th>VALUE</th>
<th>MEANING</th>
<th>FIELD</th>
</tr>
</thead>
<tbody>
<tr>
<td>STSSV_CODE</td>
<td>3</td>
<td>position of 14:3</td>
<td></td>
</tr>
<tr>
<td>STSSS_CODE</td>
<td>12</td>
<td>size of 14:3</td>
<td>message code</td>
</tr>
<tr>
<td>STSSM_CODE</td>
<td>mask</td>
<td>mask for 14:3</td>
<td></td>
</tr>
<tr>
<td>STSSV_SEVERITY</td>
<td>0</td>
<td>position of 2:0</td>
<td></td>
</tr>
<tr>
<td>STSSS_SEVERITY</td>
<td>3</td>
<td>size of 2:0</td>
<td>severity</td>
</tr>
<tr>
<td>STSSM_SEVERITY</td>
<td>7</td>
<td>mask for 2:0</td>
<td></td>
</tr>
<tr>
<td>STSSV_SUCCESS</td>
<td>0</td>
<td>position of 0</td>
<td></td>
</tr>
<tr>
<td>STSSS_SUCCESS</td>
<td>1</td>
<td>size of 0</td>
<td>success</td>
</tr>
<tr>
<td>STSSM_SUCCESS</td>
<td>1</td>
<td>mask for 0</td>
<td></td>
</tr>
</tbody>
</table>

Interpretation of Severity Codes

A severity code of 1 indicates that the procedures 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 3 indicates that the procedure generating the condition value was successfully completed, but has some parenthetical information included in a message if the condition were to be signaled.

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 severe-errors as failures. If the condition value is signalled, 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, information and successes as the basis for continuing.

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

<table>
<thead>
<tr>
<th>SEVERITY</th>
<th>ROUTINE</th>
<th>SIGNAL</th>
<th>DEFAULT AT PROGRAM EXIT</th>
</tr>
</thead>
<tbody>
<tr>
<td>success</td>
<td>normal</td>
<td>continue</td>
<td>continue</td>
</tr>
<tr>
<td>information</td>
<td>normal</td>
<td>continue</td>
<td>continue</td>
</tr>
<tr>
<td>warning</td>
<td>failure</td>
<td>continue</td>
<td>continue</td>
</tr>
<tr>
<td>error</td>
<td>failure</td>
<td>continue</td>
<td>stop job</td>
</tr>
<tr>
<td>severe-error</td>
<td>failure</td>
<td>exit</td>
<td>stop job</td>
</tr>
</tbody>
</table>

The default for signaled messages is to output a message to file SYS$OUTPUT. In addition, for severities other than success (STS$_SUCCESS) a copy of the message is made on file SYS$ERROR. At program exit, success and information completion values do not generate messages, while warning, error, and severe-error condition values generate messages to both files, SYS$OUTPUT and SYS$ERROR, unless bit 28 (STS$V_INHIB_MSG) is set.

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

Use of Condition Values
VAX-11 software components 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 interpreter. User-generated values should set bit 27 and bit 15 so that these condition values will not conflict with values generated by DIGITAL.

REGISTER USAGE
The following registers have defined uses:

<table>
<thead>
<tr>
<th>Register</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>PC</td>
<td>program counter</td>
</tr>
<tr>
<td>SP</td>
<td>stack pointer</td>
</tr>
<tr>
<td>FP</td>
<td>current stack frame pointer. Must always point at current frame; no modification permitted within a procedure body.</td>
</tr>
<tr>
<td>AP</td>
<td>Argument Pointer. When a call occurs, AP must point to a valid argument list. A procedure without parameters points to an argument list consisting of a single longword containing the value 0.</td>
</tr>
</tbody>
</table>
Appendix C

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

All other registers except R0, R1 are preserved across procedure calls. The called procedure can use any of them if it saves and restores registers R2 through R11, using the procedure entry mask mechanism. It may use R0 and R1, and should not specify them in the entry mask. The entry mask mechanism must be used so that any stack unwinding done by the condition handling mechanism will correctly restore all registers. For the same reason, if JSB routines are used, they must not save any registers not already saved by the entry mask mechanism of the calling program.

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. Any other use of FP at any time within a procedure is prohibited.

The contents of the stack located at higher addresses than the mask/PSW longword belong to the calling program; it should not be read or written by the called procedure, except as specified in the argument list. The contents 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.

The called procedure allocates stack storage by subtracting the required number of bytes from the SP provided on entry. This local storage is automatically freed by the RET instruction.

Bit 28 of the mask/PSW longword is reserved to DIGITAL for future extensions to the stack frame.
ARGUMENT DATA TYPES
The following encoding is used for data elements:

<table>
<thead>
<tr>
<th>SYMBOL</th>
<th>CODE</th>
<th>DESCRIPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>DSC$K_DTYPE_Z</td>
<td>0</td>
<td>Unspecified; the calling program has specified no data type; the called procedure should assume the argument is of the correct type.</td>
</tr>
<tr>
<td>DSC$K_DTYPE_V</td>
<td>1</td>
<td>Bit; ordinarily a bit string.</td>
</tr>
<tr>
<td>DSC$K_DTYPE_BU</td>
<td>2</td>
<td>Byte Logical; 8-bit unsigned quantity.</td>
</tr>
<tr>
<td>DSC$K_DTYPE_WU</td>
<td>3</td>
<td>Word Logical; 16-bit unsigned quantity.</td>
</tr>
<tr>
<td>DSC$K_DTYPE_LU</td>
<td>4</td>
<td>Longword Logical; 32-bit unsigned quantity.</td>
</tr>
<tr>
<td>DSC$K_DTYPE_QU</td>
<td>5</td>
<td>Quadword Logical; 64-bit unsigned quantity.</td>
</tr>
<tr>
<td>DSC$K_DTYPE_B</td>
<td>6</td>
<td>Byte Integer; 8-bit signed 2’s-complement integer.</td>
</tr>
<tr>
<td>DSC$K_DTYPE_W</td>
<td>7</td>
<td>Word Integer; 16-bit signed 2’s-complement integer.</td>
</tr>
<tr>
<td>DSC$K_DTYPE_L</td>
<td>8</td>
<td>Longword Integer; 32-bit signed 2’s-complement integer.</td>
</tr>
<tr>
<td>DSC$K_DTYPE_Q</td>
<td>9</td>
<td>Quadword Integer; 64-bit signed 2’s-complement integer.</td>
</tr>
<tr>
<td>DSC$K_DTYPE_F</td>
<td>10</td>
<td>Single-precision Floating; 32-bit VAX-11 floating point.</td>
</tr>
<tr>
<td>DSC$K_DTYPE_D</td>
<td>11</td>
<td>Double-precision Floating; 64-bit VAX-11 floating point.</td>
</tr>
<tr>
<td>DSC$K_DTYPE_FC</td>
<td>12</td>
<td>Complex; ordered pair of single-precision floating-point quantities, representing a complex number. The lower addressed quantity represents the real part, the higher addressed represents the imaginary part.</td>
</tr>
</tbody>
</table>
Appendix C

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:

<table>
<thead>
<tr>
<th>SYMBOL</th>
<th>CODE</th>
<th>DESCRIPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>DSC$K_DTYPE_T</td>
<td>14</td>
<td>ASCII text string; a sequence of 8-bit ASCII characters.</td>
</tr>
<tr>
<td>DSC$K_DTYPE_NU</td>
<td>15</td>
<td>Numeric string, unsigned.</td>
</tr>
<tr>
<td>DSC$K_DTYPE_NL</td>
<td>16</td>
<td>Numeric string, left separate sign.</td>
</tr>
<tr>
<td>DSC$K_DTYPE_NLO</td>
<td>17</td>
<td>Numeric string, left overpunched sign.</td>
</tr>
<tr>
<td>DSC$K_DTYPE_NR</td>
<td>18</td>
<td>Numeric string, right separate sign.</td>
</tr>
<tr>
<td>DSC$K_DTYPE_NRO</td>
<td>19</td>
<td>Numeric string, right overpunched sign.</td>
</tr>
<tr>
<td>DSC$K_DTYPE_NZ</td>
<td>20</td>
<td>Numeric string, zoned sign.</td>
</tr>
<tr>
<td>DSC$K_DTYPE_P</td>
<td>21</td>
<td>Packed decimal string.</td>
</tr>
<tr>
<td>DSC$K_DTYPE_ZI</td>
<td>22</td>
<td>Sequence of instructions.</td>
</tr>
<tr>
<td>DSC$K_DTYPE_ZEM</td>
<td>23</td>
<td>Procedure entry mask.</td>
</tr>
</tbody>
</table>

The following types codes are reserved for future use:

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

ARGUMENT DESCRIPTORS
A uniform descriptor mechanism is defined for use by all procedures that conform to the VAX-11 calling standard. Descriptors are uniformly typed and the mechanism is extensible. Each class of descriptor consists of at least two longwords in the following format:
Appendix C

<table>
<thead>
<tr>
<th>CLASS</th>
<th>DTYPE</th>
<th>LENGTH</th>
<th>DESCRIPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>:DESCRIPTOR</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>POINTER</td>
</tr>
</tbody>
</table>

**SYMBOL**

**DSC$W_LENGTH**

\(<0,15:0>\)

A one-word field specific to the descriptor class; typically a 16-bit (unsigned) length.

**DSC$B_DTYPE**

\(<0,23:16>\)

A one-byte data type code.

**DSC$B_CLASS**

\(<0,31:24>\)

A one-byte descriptor class code.

**DSC$A_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 can be 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; thus, no more than the above information can be assumed.

**Scalar and String Decriptors**

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

<table>
<thead>
<tr>
<th>I</th>
<th>DTYPE</th>
<th>LENGTH</th>
<th>DESCRIPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>:DESCRIPTOR</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>POINTER</td>
</tr>
</tbody>
</table>

**SYMBOL**

**DSC$W_LENGTH**

Length of data item in bytes, unless the DSC$B_TYPE field contains the value 1 (bit) or 21 (packed decimal). Length of data item is in bits for bit string. Length of data item is in digits (4 bits) for packed string.

361
Appendix C

DSC$B_DTYPE
DSC$B_CLASS
DSC$A_POINTER

A one-byte data type code.
1 = DSC$K_CLASS_S
Address of first byte of data storage.

If the string must be extended in a string comparison or is being copied to a fixed length string containing a greater length, the ASCII space character (hex 20) is used as the fill character.

Dynamic String Descriptor
A single descriptor form is used for dynamically allocated strings.

<table>
<thead>
<tr>
<th></th>
<th>DTYPE</th>
<th>LENGTH</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>SYMBOL</th>
<th>DESCRIPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>DSC$W_LENGTH</td>
<td>Length of data item in bytes, unless the DSC$B_TYPE field contains the value 1 (bit) or 21 (packed decimal). Length of data item is in bits for bit string. Length of data item is in digits (4 bits) for packed string.</td>
</tr>
<tr>
<td>DSC$B_DTYPE</td>
<td>A one-byte data type code.</td>
</tr>
<tr>
<td>DSC$B_CLASS</td>
<td>2 = DSC$K_CLASS_D</td>
</tr>
<tr>
<td>DSC$A_POINTER</td>
<td>Address of first byte of data storage.</td>
</tr>
</tbody>
</table>

When a string is written, the length field and/or the pointer field may be changed. The VAX-11 Run-Time Library provides procedures for allocating, copying and deallocating dynamic strings.

Varying String Descriptor
Reserved to DIGITAL.

362
Array Descriptor
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:

<table>
<thead>
<tr>
<th>4</th>
<th>DTYPE</th>
<th>LENGTH</th>
<th>: DESCRIPTOR</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>BLOCK 1 - PROTOTYPE</td>
</tr>
<tr>
<td>DMICT</td>
<td>AFLAGS</td>
<td>RESERVED</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ARSIZE</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

|       |       |       | BLOCK 2 - MULTIPLIERS |
|       |       |       | M1    |
|       |       |       | ...   |
|       |       |       | M(n-1) |
|       |       |       | Mn    |

|       |       |       | BLOCK 3 - BOUNDS |
|       |       |       | L1    |
|       |       |       | U1    |
|       |       |       | ...   |
|       |       |       | Ln    |
|       |       |       | Un    |

**SYMBOL**

**DESCRIPTION**

DSC$W_LENGTH
Data element size (in bytes unless the DSC$B_TYPE field contains the value 1 or 21)

DSC$B_DTYPE
A one-byte data type code.
4 = DSC$K_CLASS_A

DSC$B_CLASS
Address of first actual byte of data storage.

DSC$A_POINTER
Reserved

<2,15:0>

Reserved for future use (MBZ).

<2,23:16>

Array flag bits.

<2,20:16>

MBZ

363
Appendix C

DSC$V_FL_COLUMN <2,21>
If set, the elements of the array are stored by columns (FORTRAN). Otherwise the elements are stored by rows.

DSC$V_FL_COEFF <2,22>
If set, the multiplicative coefficients in Block 2 are present. Must be set if DSC$V_FL_BOUNDS is set.

DSC$V_FL_BOUNDS <2,23>
If set, the bounds information in Block 3 is present. Requires that DSC$V_FL_COEFF be set.

DSC$B_DIMCT <2,31:24>
Number of dimensions

DSC$L_ARSIZE <3,31:0>
Total size of array (in bytes unless the DSC$B_TYPE field contains the value 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 zero-origin arrays.

DSC$L_Mi <4+i, 31:0>
Addressing coefficients (Mi = U_i - 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.

Procedure Descriptor
The descriptor for a procedure specifies its entry address and function value data type, if any. A procedure descriptor has the form:

<table>
<thead>
<tr>
<th>5</th>
<th>DTYPE</th>
<th>LENGTH</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>POINTER</td>
</tr>
</tbody>
</table>

364
Appendix C

DSC$W_LENGTH
DSC$B_DTYPE
DSC$B_CLASS
DSC$A_POINTER

Length associated with the function value.
Function value data type.
5 = DSK$K_CLASS_P
Address of entry mask to routine.

Procedures return values in R0 or R0 and R1 as follows:
1. If the return value is a scalar, then the value is in R0 or R0/R1. The
data type and length are specified as DSC$B_DTYPE and
DSC$W_LENGTH in the procedure descriptor.
2. If the return value is not a scalar (i.e., if an array, a string, a
procedure, etc.), no function value can be returned. Instead, the
argument expressed as a function value is passed as the first
argument and the other arguments are shifted down by one.

Procedure Incarnation Descriptor
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. A procedure incarnation
descriptor has the form:

<table>
<thead>
<tr>
<th>6</th>
<th>DTYPE</th>
<th>LENGTH</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>POINTER</td>
<td></td>
</tr>
<tr>
<td></td>
<td>FRAME ADDRESS</td>
<td></td>
</tr>
</tbody>
</table>

SYMBOL
DSC$W_LENGTH
DSC$B_DTYPE
DSC$B_CLASS
DSC$A_POINTER
DSC$A_FRAME
<2,31:0>

DESCRIPTION
Length associated with the function value
Function value data type code
6 = DSK$K_CLASS_PI
Address of entry mask to routine.
Address of frame of this incarnation.

Label Descriptor
The descriptor for a label specifies the start of its code. It has the
following form:

<table>
<thead>
<tr>
<th>7</th>
<th>DTYPE</th>
<th>LENGTH</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>POINTER</td>
<td></td>
</tr>
</tbody>
</table>

365
Appendix C

**SYMBOL**

DSC$W_LENGTH
DSC$B_DTYPE
DSC$B_CLASS
DSC$A_POINTER

**DESCRIPTION**

Not used; MBZ.
Not used; MBZ.
7 = DSC$K_CLASS_J.
Address of label to jump to.

**Label Incarnation Descriptor**
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 pro-
cedure. The label incarnation descriptor has the following form:

<table>
<thead>
<tr>
<th>SYMBOL</th>
<th>DESCRIPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>DSC$W_LENGTH</td>
<td>Not used; MBZ.</td>
</tr>
<tr>
<td>DSC$B_DTYPE</td>
<td>Not used; MBZ.</td>
</tr>
<tr>
<td>DSC$B_CLASS</td>
<td>8 = DSC$K_CLASS_J.</td>
</tr>
<tr>
<td>DSC$A_POINTER</td>
<td>Address of label to jump to.</td>
</tr>
</tbody>
</table>
| DSC$A_FRAME     | Address of frame of this incarna-
| <2,31:0>        | tion.                       |

**Reserved Descriptors**
Descriptor classes 9-191 are reserved to DIGITAL. Classes 192
through 255 are reserved to DIGITAL’s Computer Special Systems
Group and customers.

**VAX-11 CONDITIONS**
A condition is either 1) a hardware generated synchronous exception,
or 2) a software event that is to be processed in a manner analogous to
a hardware exception. Floating overflow trap, memory access viola-
tion exception, and the reserved operation exception are examples of
hardware generated conditions. An output conversion error, an end-
of-file, or the filling of an output buffer are examples of software events
that might be treated as conditions.

Depending on the condition and on the program, four types of action
may be taken when a condition occurs.
Appendix C

1. Ignore the condition. For example, if an underflow occurs in a floating point operation, continuing from the point of the exception with a zero result may be satisfactory.

2. Take some special action and then continue from the point where the condition occurred. For example, if the end of a buffer is reached while a series of data items are being written, the special action is to start a new buffer.

3. Terminate the operation and branch from the sequential flow of control. For example, if the end of an input file is reached, the branch exits from a loop that is processing the input data.

4. Treat the condition as an unrecoverable error. For example, when the floating divide by zero exception condition occurs, the program exits after writing (optionally) an appropriate error message.

When an unusual event or error occurs in a called procedure, the procedure can return a condition value to the caller indicating what has happened. 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.

Condition Handlers
To handle hardware- or software-detected exceptions, the VAX-11 Condition Handling Facility allows the programmer to specify a condition handler procedure to be called when an exception condition occurs. This same handler procedure may also be used to handle software-detected conditions.

An active procedure may establish a condition handler to be associated with it. The presence of a condition handler is indicated by a non-zero address in the first longword of the procedure’s stack frame. When an event occurs that is to be treated using the condition handling facility, the procedure detecting the event signals the event by calling the facility and passing a condition value describing the condition that occurred. This condition value has the same format and interpretation as a function value. All hardware exceptions are signaled.

When a condition is signaled, the condition handling facility looks for a condition handler in the current procedure’s stack frame. If a handler is found it is entered. If no handler is associated with the current procedure, the immediately preceding stack frame is examined.
Again, if a handler is found it is entered. If a handler is not found the search of previous stack frames continues until the default condition handler established by the system is reached or the stack runs out. The default condition handler prints messages indicated by the signal argument list by calling the Put Message (SYS$PUTMSG) system service, followed by an optional symbolic stack traceback. Success conditions with STS$K_SUCCESS result in messages to file SYS$OUTPUT only. All other errors, including informational messages (STS$K_INFO), produce messages on files SYS$OUTPUT and SYS$ERROR.

For example, if a procedure needs to keep track of the occurrence of the floating-point underflow exception, it can establish a condition handler to examine the condition value passed when the handler is invoked. Then when the floating-point underflow exception occurs, the condition handler will be entered and will log the condition. The handler will return to the instruction immediately following the instruction causing the underflow.

If floating point operations occur in many procedures of a program, the condition handler can be associated with the program's main procedure. When the condition is signalled, successive stack frames are 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.

**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, 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 preceding stack frame.
3. Signal a different condition.
4. Unwind the stack.

OPERATIONS INVOLVING CONDITION HANDLERS
The functions provided by the VAX-11 Condition Handling Facility are to:

1. Establish a condition handler. A condition handler is associated with the current procedure by placing the handler's address in the current procedure's activation stack frame.

2. Revert to the caller's handling. If a condition handler has been established, it can be removed by clearing its address in the current procedure activation's stack frame.

3. Enable or disable certain arithmetic exceptions. The following hardware exceptions can be enabled or disabled by software: floating-point underflow, integer overflow, and decimal overflow. No signal occurs when the exception has been disabled.

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 frames occurring before the signal from the stack. During the unwinding operation, the stack is scanned, and if a condition handler is associated with a frame, that handler is entered before the frame is removed. Unwinding the stack allows a procedure to perform application-specific cleanup operations before exiting.

Establish A Condition Handler
Each procedure activation has a condition handler potentially attached to it using longword 0 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 three statically allocated 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, and for the system to establish a last-chance handler. Since these handlers do not obey the procedure nesting rules, they should not be used by modular code. Instead the stack based declaration should be used.

The code to establish a condition handler is:

```
MOVAL     handler-entry-point, 0(PF)
```
Revert to the Caller’s Handling
Reverting to the caller’s handling 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 to the caller’s handling is:

`CLRL 0(FP)`

Signal a Condition
The signal operation is the method used for indicating that an exception condition has occurred. To issue a message and potentially continue execution after handling the condition, a program calls the LIB$SIGNAL procedure as follows:

`CALL LIB$SIGNAL (condition-value, arg-list...)`

To issue a message, but not continue execution, a program calls LIB$STOP, as follows:

`CALL LIB$STOP (condition-value, arg-list...)`

In both cases, condition-value indicates the condition that is being signaled. However, LIB$STOP sets the severity of the condition-value to be a severe-error. The remaining arguments describe the details of the exception. These are the same arguments used to issue a system message. Note that unlike most calls, LIB$SIGNAL and LIB$STOP preserve R0 and R1, as well as the other registers. Therefore, a debugger can insert a call to LIB$SIGNAL to display the entire state of the process at the time of the exception. It also allows signals to be coded in VAX-11 MACRO without changing the register usage. This is useful for debugging checks and gathering statistics. 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 and finally the last-chance exception vector, if necessary. The current and previous stack frames are found by using FP and chaining back through the stack frames using the saved FP in each frame. The exception vectors have three address locations per access mode.

As a part of image start-up, the system declares a default last-chance handler. This handler is used as a last resort when the normal handlers are not performing correctly. The debugger can replace the default system last-chance handler with its own.

In some frame before the call to the main program, the system establishes a default catch-all condition handler that issues system messages. In a subsequent frame before the call to the main program, the system usually establishes a traceback handler. These system-sup-
plied condition handlers use condition-value to get the message and then use the remainder of the argument list to format and output the message through the Put Message SYS$PUTMSG system service. If the severity field of the condition value (bits 0 through 2) does not indicate a severe-error (that is, a value of 4), these default condition handlers return with SS$.CONTINUE. If the severity is severe-error, these default handlers exit the program image with the condition value as the final image status.

The stack search terminates when the old FP is 0 or is not accessible or when 64K frames have been examined. If no condition handler is found, or all handlers returned with SS$.RESIGNAL, then the vectored last-chance handler is called.

If a handler returns SS$.CONTINUE, and LIB$STOP was not called, control returns to the signaler. Otherwise, LIB$STOP issues a message that an attempt was made to continue from a noncontinuable exception and exits with the condition value as the final image status.

<table>
<thead>
<tr>
<th>Call to:</th>
<th>Signed Condition</th>
<th>Default Handler</th>
<th>Handler Specifies Continue</th>
<th>Handler Specifies UNWIND</th>
<th>No Handler Is Found (Stack Bad)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>&lt;2:0&gt;</td>
<td>Gets Control</td>
<td>RET</td>
<td>UNWIND</td>
<td>Call last chance handler EXIT</td>
</tr>
<tr>
<td></td>
<td>&lt;4</td>
<td>condition message RET</td>
<td>RET</td>
<td>UNWIND</td>
<td>Call last chance handler EXIT</td>
</tr>
<tr>
<td>LIB$SIGNAL</td>
<td>=4</td>
<td>condition message EXIT</td>
<td>RET</td>
<td>UNWIND</td>
<td>Call last chance handler EXIT</td>
</tr>
<tr>
<td>or hardware exception</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LIB$STOP</td>
<td>force condition =4</td>
<td>message EXIT</td>
<td>“cannot continue” EXIT</td>
<td>UNWIND</td>
<td>Call last chance handler EXIT</td>
</tr>
</tbody>
</table>

371
PROPERTIES OF CONDITION HANDLERS

Condition Handler Parameters and Invocation
If a condition handler is found on a software detected exception, the handler is called with an argument list consisting of:

\[
\text{continue} = \text{handler} (\text{signal-args}, \text{mechanism-args})
\]

Each argument is a reference to a longword vector. The first longword of each vector is the number of remaining longwords in the vector. The symbols CHF$\_L\_SIGARGGLST (=4) and CHF$\_L\_MCHARGLST (=8) can be used to access the condition handler arguments relative to AP.

Signal-args is the condition argument list from the call to LIB$SIGAL or LIB$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 through 0 of the condition-value indicate severity and do not indicate which condition is being signalled, the handler should examine only the condition identifi-
cation, i.e., condition-value (bit 31 through 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

<table>
<thead>
<tr>
<th>4</th>
<th>CHF$_L_MCH_ARGS</th>
</tr>
</thead>
<tbody>
<tr>
<td>FRAME</td>
<td>CHF$_L_MCH_FRAME</td>
</tr>
<tr>
<td>DEPTH</td>
<td>CHF$_L_MCH_DEPT</td>
</tr>
<tr>
<td>RO</td>
<td>CHF$_L_MCH_SAVR0</td>
</tr>
<tr>
<td>R1</td>
<td>CHF$_L_MCH_SAVR1</td>
</tr>
</tbody>
</table>

The frame is the contents of FP in the establisher’s context. This can be used as a base to access the local storage of the establisher if the restrictions in the Use of Memory section are met. The depth is a positive count of the number of procedure activation stack frames between the frame in which the exception occurred and the frame depth that established the handler being called. Depth has the value 0 for an exception handled by the procedure activation invoking the exception (that is, containing the instruction causing the hardware exception or calling LIB$SIGAL). Depth has positive values for pro-
cedure activations calling the one having the exception (1 for the im-
mediate caller, etc.). If a system service gives an exception, the immediate caller of the service is notified at depth = 1. Depth has value −2 when the condition handler is established by the primary exception vector, −1 when it is established by the secondary vector, and −3 when it is established by the last-chance vector. The contents of R0 and R1 are the same as at the time of the call to LIB$SIGNAL or LIB$STOP.

For hardware detected exceptions, the condition-value indicates which exception vector was taken and the next 0 or several longwords are additional parameters. The remaining two longwords are the PC and PSL:

<table>
<thead>
<tr>
<th>n</th>
<th>CHFSL_SIG_ARGS</th>
</tr>
</thead>
<tbody>
<tr>
<td>CONDITION_VALUE</td>
<td>CHFSL_SIG_NAME</td>
</tr>
<tr>
<td>NONE OR SOME ADDITIONAL ARGUMENTS</td>
<td></td>
</tr>
<tr>
<td>PC</td>
<td></td>
</tr>
<tr>
<td>PSL</td>
<td></td>
</tr>
</tbody>
</table>

If one of the default condition handlers established by the system is entered, it calls the Put Message SYS$PUTMSG system service, to interpret the signal argument list and output the indicated information or error message. See the description of SYS$PUTMSG in the VAX/VMS Systems Services Reference Manual for the format of the signal argument list.

**Use of Memory**

A condition handler and all procedures which it calls are restricted to referring to explicitly passed arguments only. Handlers cannot refer to common or other external storage and they cannot reference local storage in the procedure that established the handler. The existence of handlers thus does not affect compiler optimization. Compilers that do not follow this rule must ensure that any variables referred to by the handler are always kept in memory.

**Returning From a Condition Handler**

Condition handlers are invoked by the VAX-11 Condition Handling Facility. Therefore, the return from the condition handler is to the condition handling facility.
Appendix C

To continue from the instruction following the signal, the handler must return with the function value SS$\_CONTINUE ("true," that is, with bit 0 set). If, however, the condition was signaled with a call to LIB$STOP, the image will exit. To resignal the condition, the condition handler returns with the function value SS$\_RESPAWN ("false," that is, with bit 0 clear). To alter the severity of the signal, the handler modifies the low three bits of the condition-value longword in the signal-args vector and resinals. If the condition handler wants to alter the defined control bits of the signal, the handler modifies bits 31:28 of condition-value and resinals. To unwind, the handler calls SYS$UNWIND and then returns. In this case the handler function value is ignored.

Request to Unwind
To unwind, the handler or any procedure it calls can perform:

\[
\text{success = SYS$UNWIND}
\]
\[
([\text{depadr = handler depth + 1},
\text{[new-PC = return PC]}])
\]

The argument depadr specifies the address of a longword that contains the number of pre-signal frames (depth) to remove. If that number is less than or equal to 0, then nothing is to be unwound. The default (address=0) is to return from the estabisher of the handler. To unwind to the estabisher, the depth from the call to the handler should be specified. When the handler is at depth 0, it can achieve the equivalent of an UNWIND operation to an arbitrary place in its estabisher by altering the PC in its signal-args vector and returning with SS$\_CONTINUE instead of performing an UNWIND.

The argument new-PC specifies the location to receive control when the unwinding operation is complete. The default is to continue at the instruction following the call to the last procedure activitation removed from the stack.

The function value SUCCESS is a standard success code (SS$\_NORMAL), or indicates failure with one of the following return status condition values:

- No signal active (SS$\_NOSIGNAL)
- Alrearding unwinding (SS$\_UNWINDING)
- Insufficient frames for depth (SS$\_INSFRAME)

The unwinding operation occurs when the handler returns to the condition handling facility. Unwinding is done by scanning back through the stack and calling each handler that has been associated with a frame. The handler is called with exception of SS$\_UNWIND to perform any application-specific cleanup. In particular, if the depth speci-
fied includes unwinding the establisher’s frame, the current handler will be recalled with this unwind exception.

The call to the handler takes the same form as 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. To specify the value of the top level “function” being unwound, the handler should modify R0 and R1 in the mechanism-args vector. They will then be restored from the mechanism-args vector at the end of the unwind.

Depending on the arguments to SYS$UNWIND, the unwinding operation will be terminated as follows:

- SYS$UNWIND (,)—unwind to the establisher’s caller with establisher function value from R0 and R1 in the mechanism vector.
- SYS$UNWIND (depth, )—unwind to the establisher at the point of the call to LIB$SIGNAL or LIB$STOP or the instruction that resulted in the exception. The contents of R0 and R1 are restored from R0 and R1 in the mechanism-args vector.
- SYS$UNWIND (depth, location)—unwind to a specified activation and transfer to the specified location with the contents of R0 and R1 from R0 and R1 in the mechanism-args vector.

SYS$UNWIND can be called whether the condition was a software exception signaled by calling LIB$SIGNAL or LIB$STOP, or was a hardware exception. Calling SYS$UNWIND is the only way to continue execution after a call to LIB$STOP.

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. To find the registers R2 through FP, a scan up the stack frames must be performed, starting with the current frame and ending with up to and including the call to the handler. During the scan, the last frame found to save a register contains that register's contents at the time of the exception. If no frame saved the register, the register is still active in the current procedure. The frame of the call to the handler can be identified by the return address of SYS$CALL_HANDL+4. Thus the registers are:

- R0, R1: In mechanism-args
- R2..R11: Last frame saving it
- AP: old AP of SYS$CALL_HANDL+4 frame
- FP: old FP of SYS$CALL_HANDL+4 frame
- SP: equal to end of signal-args
- PC, PSL: at end of signal-args

**MULTIPLE ACTIVE SIGNALS**

A signal is said to be active until the signaller gets control again or is unwound. A signal can occur while a condition handler or a procedure it has called is executing in response to a previous signal. For example, procedure (A, B, C ...) establishes a condition handler (Ah, Bh, Ch, ...). If A calls B calls C which signals S and Ch resignals, then Bh gets control. If Bh calls procedure X and X calls procedure Y and Y signals T, the stack is:

```
<signal T>
  Y
  X
  Bh
<signal S>
  C
  B
  A
```

which was programmed:

```
A
  B ---X---Bh
   C
<signal S>
  Y
<signal T>
```

The handlers are searched for in the order: Yh, Xh, <Bh>h, Ah. Note that Ch is not called because it is a structural descendant of B. Bh is not called again because that would require it to be recursive. Recursive handlers could not be coded in nonrecursive languages.
such as FORTRAN. Instead Bh can establish itself or another procedure as its handler (Bhh).

The following algorithm is used on the second and subsequent signals which occur before the handler for the original signal returns to the condition handling facility. The primary and secondary exception vectors are checked. Then, however, the search backward in the process stack is modified. In effect, the stack frames traversed in the first search are skipped over in the second search. Thus, the stack frame preceding the first condition handler up to and including the frame of the procedure that has established the handler is skipped. Despite this skipping, depth is not incremented. The stack frames traversed in the first and second search are skipped over a third search, etc. Note that if a condition handler signals, it will not automatically be invoked recursively. However, if a handler itself establishes a handler, this second handler will be invoked. Thus, a recursive condition handler should start by establishing itself. Any procedures invoked by the handler are treated in the normal way; that is, exception signaling follows the stack up to the condition handler.

If an unwinding operation is requested while multiple signals are active, all the intermediate handlers are called for the operation. For example, in the above diagram, if Ah specifies unwinding to A, the following handlers will be called for in the unwind: Yh, Xh, Bhh, Ch, and Bh.

For proper hierachical operation, an exception that occurs during execution of a condition handler established in an exception vector should be handled by that handler rather than propagating up the activation stack. The vectored condition handler should establish a handler in its stack frame to handle all exceptions.
APPENDIX D
PROGRAMMING EXAMPLES

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.

SORT ALGORITHM
The following subroutine written in FORTRAN is an algorithm for sorting an array of values into ascending order.

```fortran
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
```

379
8. MOVx (R0) [R1], R3 ;Save A (I)
9. MOVx (R0) [R2], (R0) [R1] ;Replace A (I) with A (J)
10. MOVx R3, (R0) [R2] ;Replace A (J) with saved A (I)
11. 10$: AOBLEQ R12, R2, 2$ ;Continue
12. AOBLSs R12, R1, 1$ ;Continue
13. RET ;Return and restore ;registers R2 and R3

Line 1 contains an entry mask so that registers R2 and R3 will be saved by the CALL instruction which calls the subroutine. By convention, R0 and R1 are not saved. Integer overflow is enabled.

Line 2 gets the base of the A array. The move address instruction is used in conjunction with argument mode addressing. This instruction saves memory accesses inside the loop.

Line 3 gets the array size. The move long instruction is used in conjunction with argument mode addressing. This instruction saves memory accesses inside the loop.

Line 4 initializes I to 1. Literal mode addressing is used.

Line 5 initializes J with I + 1. A three operand add is used.

Line 6 compares A (I) to A (J). Register post-indexed mode addressing is used.

Line 7 branches past the exchange if the array elements are in the right order.

Lines 8 through 10 exchange the array elements if they are in the wrong order. Register post-indexed mode addressing is used.

Lines 11 and 12 carry out the loop end operations. Argument mode addressing is used.

Line 13 returns and restores registers R2 and R3.

Note, that because of logical indexing in Lines 5, 7, 8, and 9 and the orthogonality of operator and data type, the subroutine words for byte, word, longword, floating or double data types of array A simply by substituting B, W, L, F, or D respectively for x. Note that if double, then R4 would have to be saved also in the entry mask.
Appendix D

The size of each instruction is:

1. 2 bytes
2. 4
3. 4
4. 3
5. 4
6. 5
7. 2
8. 4
9. 5
10. 4
11. 4
12. 4
13. 1
Total 46 bytes

SIN FUNCTION
This example shows how the initial argument handling might be done in the math library to handle argument range reduction followed by CASEing to the algorithm for each octant.

; ; X = SIN (Y)

PIHI = xxx ; high 4 bytes (8 if double)
PILO = xxx ; low byte of 4/PI

SIN::

.WORD $400C ; save R2-R3 for POLYF, —R7 for POLYD
MOVAL HANDLER, 0 (FP) ; enable integer overflow
; condition handler to catch ; loss of significance on ; a huge argument
EMODx #PIHI, #PILO, @4 (AP), R2, R0 ; get octant in R2
; reduced argument in R0
BGEQ 1$ ; if positive, ok
ADDx #FL. 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

381
Appendix D

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$, 1$
RET

;evaluate polynomial
;return value in R0

1$:.FLOAT ...

... .FLOAT ...

... 2$ = .-1$-1

... HANDLER:

;condition handler

.WORD ...

... FIXED FORMAT FLOATING OUTPUT
This example shows how to output a floating point number in the FORTRAN format F9.3.

;string = FOUT (X)

;STRING: .BLKB 10

;room for output

PATTERN;

EO$FLOAT 4
EO$END_FLOAT
EO$MOVE 1
EO$INSERT ↑A./
EO$MOVE 3
EO$END

;EDITPC pattern string
;float sign, move 4 digits
;end floating sign
;move one digit
;insert period
;move three fractional digits
;end of pattern
Appendix D

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

COBOL OUTPUT EDITING
In all of these examples, A is a COMP-3 datum of length A_LEN. The operation is

MOVE A TO B.

The generated code is

EDITPC #A_LEN,@,MICRO, @B

In the patterns, the EO$ADJUST_INPUT can be omitted if A is the same size as B, and the EO$REPLACE_SIGN (and its EO$LOAD_FILL) can be omitted if A cannot contain a −0.

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

383
Appendix D

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
Appendix D

5. PICTURE

    MICRO:
    EO$ADJUST_INPUT 7
    EO$FLOAT 4
    EO$END_FLOAT 1
    EO$MOVE 2
    EO$REPLACE_SIGN 5
    EO$END

6. PICTURE

    MICRO:
    EO$ADJUST_INPUT 7
    EO$LOAD_PLUS 1
    EO$FLOAT 4
    EO$END_FLOAT 1
    EO$MOVE 2
    EO$LOAD_FILL 1
    EO$REPLACE_SIGN 5
    EO$END

7. PICTURE

    MICRO:
    EO$ADJUST_INPUT 7
    EO$LOAD_FILL 2
    EO$MOVE 2
    EO$INSERT 3
    EO$MOVE 3
    EO$SET_SIGNIF 2
    EO$INSERT 2
    EO$MOVE 3
    EO$BLANK_ZERO 2
    EO$END

8. PICTURE

    MICRO:
    EO$ADJUST_INPUT 7
    EO$FILL 3
    EO$MOVE 2
    EO$FILL 1
    EO$MOVE 3
    EO$SET_SIGNIF
EO$INSERT  
EO$MOVE  2  
EO$BLANK_ZERO  3  
EO$FILL  1  
EO$END

FORTRAN STATEMENT EVALUATION

FORTRAN
Assembly
Code:  
J = A*K + B(1)  
MOV L, R1 ;Move 1 to R1  
CVTLF K, R0 ;Convert integer K to floating point  
MULF2 A, R0 ;Multiply A*K and store in R0  
ADDF2 B ;Add B indexed by R1 to R0  
[R1], R0  
CVTFL R0, J ;Convert result in R0 to integer and store in J

This program evaluates the FORTRAN statement listed above. I is a subscript which moved to register R1. The next step of the program converts the integer K to a floating point number. Next A is multiplied by K and the result is stored in register R0. The value I, which is stored in register R1, indexes B and the calculated result is added to R0 which currently contains A*K. The last step of the program converts the floating point result back to integer format, and stores the integer in location J.

VARIABLE LENGTH FIELD

PL1 Assembly
Code: DECLARE A (1:10) BIT (5) ;Vector A, elements 1-10,  ;5 bit field  
A(l) = A(l) + 1 ;Increment lth element of  ;A and store in a
Machine Code:

INDEX I, #1, #10, #5 ;Calculate index 
#−5, R0

EXTV R0, #5, A, R1 ;Extract 5 bits and store 
;in R1

INCL R1 ;Increment R1

INSV R1, R0, #5, A ;Store 5 bits into A 
;offset by R0

This example shows the use of the variable length field instructions using the PL1 Programming Language. Its purpose is to add 1 to a particular field within a vector of fields. In the assembler code, the DECLARE statement informs the compiler that A is a vector, its elements are numbered 1 through 10, and each element is a field five bits wide. The A(I) statement increments the Ith element of A and stores the result back in A.

In the machine code, the INDEX statement consists of a lower limit of 1, an upper limit of 10, field size of 5, an offset of −5, and a temporary (R0) to store the result of the index calculation. The offset of −5 is required since the subscript starts at 1 but all indexing starts at 0.

The INDEX statement in this example checks I in the range from 1 through 10. If I is in this range it is multiplied by the field size of 5, the offset of −5 is added, and the result is stored in R0. Thus, R0 will contain the position offset of the field A(I) from the start of A. If I is outside the range 1 through 10, a subscript range trap occurs and typically results in an error message.

**LOOPS**

**FORTRAN:**

INTEGER *2 L ;Use L as a word for a DO 1 L = 3, 10, 2 ;loop counter—L is ;an integer of 2 bytes ;and loop is incremented ;by 2 for each pass ;through loop

1 CONTINUE

**Assembly Code:**

MOVW #3, L

START: ACBW #10, #2, L, START

387
FORTRAN:
INTEGER LL
DO 1 LL = 1, 10

1 CONTINUE

; Use LL as a word for a
; loop counter. Loop is
; incremented by one for
; each pass through loop.

Assembly
Code: MOVL #1, LL
START: AOBLEQ #10, LL, START
APPENDIX E
OPERAND SPECIFIER NOTATION

OPERAND SPECIFIERS
Operand specifiers are described in the following way:

\(<name> <access type> <data type>\)

where:

Name is a suggestive name for the operand in the context of the instruction. The name is often abbreviated.

Access type is a letter denoting the operand specifier access type:

a  Calculate the effective address of the specified operand. Address is returned in a longword which is the actual instruction operand. Context of address calculation is given by <data type>.

b  No operand reference. Operand specifier is a branch displacement. Size of branch displacement is given by <data type>.

m  Operand is read, potentially modified and written. Note that this is NOT an indivisible memory operation. Also note that if the operand is not actually modified, it may not be written back. However, modify type operands are always checked for both read and write accessibility.

r  Operand is read only.

v  Calculate the effective address of the specified operand. If the effective address is in memory, the address is returned in a longword which is the actual instruction operand. Context of address calculation is given by <data type>.

If the effective address is Rn, then the operand actually appears in R[n], or in R[n+1]'R[n].

w  Operand is written only.
Data type is a letter denoting the data type of the operand:

- b: byte
- d: double floating
- f: floating
- l: longword
- q: quadword
- w: word
- x: first data type specified by instruction
- y: second data type specified by instruction

**OPERATION DESCRIPTION NOTATION**
The operation of each instruction is given as a sequence of control and 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, FP, or AP: the contents of register R15, R14, R13, or R12 respectively
PSW: the contents of the processor status word
PSL: the contents of the processor status long word
(x): contents of memory location whose address is x
(x)+: contents of memory location whose address is x; x incremented by the size of operand referenced at x
-(x): x decremented by size of operand to be referenced at x; contents of memory location whose address is x
<x:y>: a modifier which delimits an extent from bit position x to bit position y inclusive
<x1,x2,...,xn>: a modifier which enumerates bits x1, x2, ..., xn
Appendix E

The conventions are used:

- Other than that caused by ( ), +, or −( ), and the advancement of PC, only operands or portions of operands appearing on the left side of assignment statements are affected.
- No operator precedence is assumed, other than that replacement (←) has the lowest precedence. Precedence is indicated explicitly by { },
- All arithmetic, logical, and relational operators are defined in the context of their operand. For example, “+” applied to floating operands means a floating add while “+” applied to byte operands is an integer byte add. Similarly, “LSS” is a floating comparison when
Appendix E

applied to floating operands while “LSS” is an integer byte comparison when applied to byte operands.

- Instruction operands are evaluated according to the operand specifier conventions. The order in which operands appear in the instruction description has no effect on the order of evaluation.

- Condition codes are in general affected on the value of actual stored results, not on “true” results (which might be generated internally to greater precision). Thus, for example, 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.
APPENDIX F

ASSEMBLER NOTATION

INTRODUCTION
The VAX-11 assembler provides, as a subset, a notation which is very similar to the PDP-11 assembler notation. The principal differences are due to the fact that the VAX-11 architecture has new addressing modes and has several length variations of modes for which the PDP-11 has only a single length. For example, the PDP-11 has displacement addressing with a single displacement size of 16 bits. VAX-11 has displacement addressing in various forms with displacements of 8, 16, and 32 bits.

In general, the programmer need not be aware of the length variations in VAX-11 addressing modes. The programmer simply writes the addressing mode in a format identical to the analogous PDP-11 addressing mode, and the assembler will choose the shortest form of addressing consistent with the state of symbol definition at assembly time. Occasionally, a programmer may wish to force a given length addressing mode. The VAX-11 assembler includes a notation for accomplishing this. (Of course, if the programmer forces a length which cannot be accommodated at assembly or link time, the assembler or linker will generate an error indication.)

NOTATION FOR GENERAL MODE ADDRESSING

Register Mode
The general notation is Rn. Since results are UNPREDICTABLE if R is PC for operands taking a single register, or if Rn is SP or PC for operands taking a pair of registers, or if Rn is AP, FP, SP, or PC for operands taking four registers, the assembler generates an error indication.

Register Deferred Mode
The general notation is (Rn). Since result are UNPREDICTABLE if Rn is PC, the assembler generates an error indication.

Autoincrement Mode
The general notation is (Rn)+. For immediate mode see B.2.8.
Appendix F

Autoincrement Deferred Mode
The general notation is @(Rn)+. For absolute addressing mode see B.2.9.

Autodecrement Mode
The general notation is -(Rn). Since results are UNDEFINED if R is PC
the assembler generates an error indication.

Displacement Mode
The general notation is D(Rn). To force a byte, word, or long displace-
ment, the notation is B↑D(Rn), W↑D(Rn), L↑D(Rn) respectively. If a gen-
eral address G is used, the assembler assembles this as D(PC) where
D = G - (updated value of PC). This latter form is termed PC-relative
addressing. If a form is forced which is shorter than the actually
needed displacement, the assembler or linker generates an error indi-
cation.

Displacement Deferred Mode
The general notation is @D(Rn). To force a byte, word, or long dis-
placement, the notation is @B↑D(Rn), @W↑D(Rn), @L↑D(Rn) respec-
tively. If a general address @G is used, the assembler assembles this
as @D(PC) where D = G - (updated value of PC). This latter form is
termed PC-relative deferred addressing. If a form is forced which is
shorter than the actually needed displacement, the assembler or link-
er generates an error indication.

Literal Mode
The general notation is #cons. This results in, depending on the value
of the cons, either immediate or literal mode. To force literal mode, the
notation is S↑#cons. To force immediate mode, the notation is 1↑#
cons. If either literal or immediate mode is used on a modify or write
operand, the assembler generates an error indication.

Absolute Addressing Mode
To force a reference to an absolute address the notation is @#
location. This is assembled as autoincrement deferred using PC.

General Addressing
When a reference to a symbol will be either absolute or PC-relative,
but the choice is to be determined by the linker, the notation is
G↑location. This is assembled as a five-byte operand. The linker
chooses either @#location or L↑D(PC) depending on whether the loca-
tion is absolute or PC-relative. This is used both for general external
references and for general references between program sections
(PSECTs).
Index Mode
The general notation is `<base operand mode>[Rx]` where `<base operand mode>` is the notation for any of the addressing modes register deferred, autoincrement (immediate), autoincrement deferred (absolute), autodecrement, displacement (PC-relative), displacement deferred (PC-relative deferred) or general addressing. Since the result is UNPREDICTABLE if a register in the base operand mode is the same as the index register (except for PC), the assembler generates an error.

**GENERAL MODE ADDRESSING SUMMARY**

<table>
<thead>
<tr>
<th>Symbolic</th>
<th>Assembled Mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. R</td>
<td>register</td>
</tr>
<tr>
<td>2. (R)</td>
<td>register deferred</td>
</tr>
<tr>
<td>3. (R)+</td>
<td>autoincrement</td>
</tr>
<tr>
<td>4. -(R)</td>
<td>autodecrement</td>
</tr>
<tr>
<td>5. D(R)</td>
<td>byte, word, or longword displacement. register deferred. default is word if D is not known</td>
</tr>
<tr>
<td>6. B↑D(R)</td>
<td>byte displacement</td>
</tr>
<tr>
<td>7. W↑D(R)</td>
<td>word displacement</td>
</tr>
<tr>
<td>8. L↑D(R)</td>
<td>longword displacement</td>
</tr>
<tr>
<td>9. G</td>
<td>byte, word, or longword displacement off PC default is longword if G is not known</td>
</tr>
<tr>
<td>10. B↑G</td>
<td>byte displacement off PC</td>
</tr>
<tr>
<td>11. W↑G</td>
<td>word displacement off PC</td>
</tr>
<tr>
<td>12. L↑G</td>
<td>longword displacement off PC</td>
</tr>
<tr>
<td>13. G↑G</td>
<td>general addressing (absolute or PC-relative)</td>
</tr>
<tr>
<td>14. #cons</td>
<td>autoincrement of PC (immediate or literal)</td>
</tr>
<tr>
<td>15. S↑#cons</td>
<td>short literal</td>
</tr>
<tr>
<td>16. l↑#cons</td>
<td>immediate</td>
</tr>
<tr>
<td>17. (R)[Rx]</td>
<td>register deferred indexed</td>
</tr>
<tr>
<td>18. (R)+[Rx]</td>
<td>autoincrement indexed</td>
</tr>
<tr>
<td>19. $cons[Rx]</td>
<td>autoincrement of PC (immediate) indexed</td>
</tr>
</tbody>
</table>
20. \text{I} \uparrow \text{cons}[\text{Rx}]
\quad \text{autoincrement of PC (immediate)indexed}

21. \text{-(R)}[\text{Rx}]
\quad \text{autodecrement indexed}

22. \text{D(R)}[\text{Rx}]
\quad \text{byte, word, or longword displacement indi-}
\quad \text{cated.}
\quad \text{register deferred indexed.}

23. \text{B} \uparrow \text{D(R)}[\text{Rx}]
\quad \text{byte displacement indexed}

24. \text{W} \uparrow \text{D(R)}[\text{Rx}]
\quad \text{word displacement indexed}

25. \text{L} \uparrow \text{D(R)}[\text{Rx}]
\quad \text{longword displacement indexed}

26. \text{G}[\text{Rx}]
\quad \text{byte, word, or longword displacement off}
\quad \text{PC indexed}

27. \text{B} \uparrow \text{G}[\text{Rx}]
\quad \text{byte displacement off PC indexed}

28. \text{W} \uparrow \text{G}[\text{Rx}]
\quad \text{word displacement off PC indexed}

29. \text{L} \uparrow \text{G}[\text{Rx}]
\quad \text{longword displacement off PC indexed}

30. \text{G} \uparrow \text{location}[\text{Rx}]
\quad \text{general (absolute or PC-relative) indexed}

31. \text{ @(R)}[\text{Rx}]
\quad \text{byte displacement deferred indexed with 0}
\quad \text{displacement}

32. \text{@}(\text{R})+[\text{Rx}]
\quad \text{autoincrement deferred indexed}

33. \text{@}(\text{#location}[	ext{Rx}])
\quad \text{autoincrement of PC (immediate) deferred}
\quad \text{indexed}

34. \text{@D(R)}[\text{Rx}]
\quad \text{byte, word, or longword displacement de-}
\quad \text{ferred indexed}

35. \text{@B} \uparrow \text{D(R)}[\text{Rx}]
\quad \text{byte displacement deferred indexed}

36. \text{@W} \uparrow \text{D(R)}[\text{Rx}]
\quad \text{word displacement deferred indexed}

37. \text{@L} \uparrow \text{D(R)}[\text{Rx}]
\quad \text{longword displacement deferred indexed}

38. \text{@G}[\text{Rx}]
\quad \text{byte, word, or longword displacement off}
\quad \text{PC deferred indexed}

39. \text{@B} \uparrow \text{G}[\text{Rx}]
\quad \text{byte displacement off PC deferred indexed}

40. \text{@W} \uparrow \text{G}[\text{Rx}]
\quad \text{word displacement off PC deferred indexed}

41. \text{@L} \uparrow \text{G}[\text{Rx}]
\quad \text{longword displacement off PC deferred in-}
\quad \text{dexed}

42. \text{@}(\text{R})
\quad \text{byte displacement deferred with 0 displace-}
\quad \text{ment}

43. \text{@}(\text{R})+
\quad \text{autoincrement deferred}
Appendix F

44. @!#location  autoincrement of PC (immediate)
45. @D(R)       byte, word, longword displacement deferred
46. @B↑D(R)     byte displacement deferred
47. @W↑D(R)     word displacement deferred
48. @L↑D(R)     longword displacement deferred
49. @G          byte, word, or longword displacement off PC deferred
50. @B↑G        byte displacement off PC deferred
51. @W↑G        word displacement off PC deferred
52. @L↑G        longword displacement off PC deferred

BRANCH DISPLACEMENT ADDRESSING
The general notation is locn, where locn is the branch address. The assembler fills in the displacement displ where displ = locn – {updated value of PC}.

GENERIC OPCODE SELECTION
As a convenience to the programmer, the assembler automatically selects from among similar instructions. This allows the programmer to write code without worrying about these distinctions.

Branch Selection
If the programmer gives BR or BSB as the mnemonic, the assembler will automatically select either BRB or BRW (BSBB or BSBW) based on the distance to the label. If the label is not yet defined, the word branch displacement form will be selected.

Number of Operand Selection
If the programmer omits the final digit from those opcodes which have two forms (e.g., ADDW instead of ADDW2 or ADDW3), the assembler will select the correct form based on the number of operands specified by the user.
The following three steps are performed in order by each instruction:

1. Each operand specifier is evaluated in order of instruction stream occurrence as follows:
   
   **access type evaluation**
   
   - **read**
     - evaluate operand location,
     - read the operand and save the operand
   
   - **write**
     - evaluate operand location
     - and save the location
   
   - **modify**
     - evaluate operand location,
     - read the operand and save both location and operand
   
   - **address,branch**
     - evaluate the address and save the address
   
   - **field**
     - evaluate operand base location and save the location

2. Perform the operation indicated by the instruction.

3. Store the result(s) using the saved address in the order indicated by the occurrence of operand specifiers in the instruction stream.

**NOTE**

The string (character and numeric) instructions write any output strings and store the registers during step 3.

The implications of this processing are:

1. Autoincrement and autodecrement operations occur as the operand specifiers are processed, and subsequent operand specifiers use the updated contents of registers modified by those operations.

**NOTE**

This implication does not necessarily apply to floating and double floating operands.
2. Except for those operations mentioned in step 1, all input operands are read and all addresses of output operands are computed before any results of the instruction are stored.

3. An operand of modify access type is not read, modified, and written as an indivisible operation. Therefore, modify access type operands cannot be used for synchronization. For synchronization refer to the ADAWI instruction (chapter 6), INSQUE and REMQUE instructions (chapter 7), and BBCCI and BBSSI instructions (chapter 8). Further information concerning synchronization can be found in the appropriate VAX-11 Hardware Handbook.

4. If an instruction references two operands of write or modify access type at the same or overlapping address, the first will be overwritten by the second. If an instruction modifies a register implicitly and also has an output operand, the output store is performed after the register update.
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 osftware 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 mode field. The operating system uses access modes to define protection levels for software executing in the context of a process. For example, the executive runs in kernel and executive mode and is most protected. The command interpreter is less protected and runs in supervisor mode. The debugger runs in user mode and is not more protected than normal users programs. 2. See record access mode.

access type  1. The way in which te processor accesses instrucion operands. Access types are: read, write, modify, address, and branch. 2. The way in which a procedure access 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 vitural 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, auto decrement, displacement, and displacement deferred. In addition, there are six indexed addressing modes using two general register, 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 index 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 actual operand of the instruction. The contents of the register are decremented according to the data type context of the register: 1 for byte, 2 for word, 4 for longword and 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 address mode, the instruction operand specifier is a signed by te 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 an 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 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 procedures reside in the control region of the process executing a compatibility mode image. The procedures intercepts 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 executed instructions.

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 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. The Current Mode field of the Processor Status Longword indicates the access mode of the currently executing software.

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 comparison is needed to locate any data in the cache because any block of main memory data can be placed in only one possible position in the cache. Contrast with fully associative cache.

displacement deferred indexed mode  An indexed addressing mode in which the base operand specifier uses displacement deferred mode addressing.

displacement deferred mode  In displacement deferred mode addressing, the specifier extension is a byte, word, or longword displacement. The displacement is sign extended to 32 bits and added to a base address obtained from the specified registers. The result is the address of a longword which contains the address of the actual operand. If the 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 contain bit 0. Bit 15 contains the sign of the number. Bits 14 through 7 contain the excess 128 binary exponent. Bits 63 through 16 and 6 through 0 contain a normalized 56-bit fraction with the redundant most significant fraction bit no 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 served operand fault. the value of a floating datum is the approximate range (+ or −) 0.29 × 10^{−38} to 1.7 × 10^{38}. The precision is approximately one part in 2^{55} or sixteen decimal digits.

**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 transmit without interpretation to the software set up to handle sequences.

**event**  A change in process status or an indication of the occurrence of some activity that concerns an individual process or cooperating processes. An incident reported to the scheduler that affects a process’ 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 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 moe 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 at 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 indentified 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 contain a normalized 24-bit fraction with the redundant most significant fraction bit no 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.

**full associative cache**  a cache organization in which any block of data from main memory ca 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.
Glossary

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 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 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 mode 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), auto-decrement 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 continuously 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, owrd, 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 in high-order for 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 sym-
bolic names for items in a data structure instead of using the numeric displacement.

**opcode** The pattern of bits within an instruction that specify the operating 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 markets, or between a marker and the end of a file.

**page fault** An exception generated by a reference to a page which is not mapped into a working set.

**page fault cluster size** The number of pages read in on a page fault.

**page frame number (PFN)** The address of the first byte of a page in physical memory. The high-order 21 bits of the physical address of the base of a page.

**page table entry (PTE)** The data structure that identifies the location and status of a page of virtual address space. When a virtual page is in memory, the PTE contains the page. When it is not in memory, the 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 possible 3-bit physical addresses that can be used to refer to locations in memory (memory space) or device register (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 addressing modes that form an effective relative to the PC.

**privileged instructions** In general any instruction intended for use by the operating system or privileged system programs. In particular, instructions that the processor will not execute unless the current access mode is kernel mode (e.g., HALT, SVPCTX, LDPCTX, MTPR, and MFPR).

**procedure** 1. A routine entered via a call instruction. 2. See command procedure.

**process** The basic entity scheduled by the system software that provides the context in which an image executes. A process consists of an address space and both hardware and software context.

**process address space** See process space.

**process context** The hardware and software contexts of a process.
**Glossary**

**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 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.
Glossary

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

read access type  An instruction or procedure operand attribute indicating that the specified operand is only ead 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 router deferred moe addressing.

register deferred mode  In register deferred mode addressing, the contents of the specified 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 super-
visor mode.

Synchronous Backplane Interconnect (SBI)  The part of the hard-
ware that interconnects the processor, memory controllers, MASS-
BUS 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 contain the
physical address of the base of the system page table.

System Control Block (SCB)  The data structure in system space
that contains all the interrupt and exception vectors known to the
system.

System Control Block Base register (SCBB)  A processor register
containing the base address of the system control block.

System Identification Register  A processor register which contains
the processor type and serial number.

System Length Register (SLR)  A processor register containing the
length of the system page table in longwords, that is, the number of
page table entries in the system region page table.

System Page Table (SPT)  The data structure that maps the system
region virtual addresses, including the addresses used to refer to the
process page tables. The system page table (SPT) contains one page table entry (PTE) for each page of system region virtual memory. The physical base address of the SPT is contained in a register called the SBR.

**system region** The third quarter of virtual address space. The lower-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 higher-addressed half of virtual address space. See also system region.

**system virtual address** A virtual address identifying a location mapped by an address in system space.

**system virtual space** See system space.

**terminal** The general name for those peripheral devices that have keyboards and video screens or printers. Under program control, a terminal enables people to type commands and data on the keyboard and receive messages on the video screen or printer. Examples of terminals are the LA36 DECwriter hard-copy terminal and VT52 video display terminal.

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

**trap** An exception condition that occurs at the end of the instruction that caused the exception. The PC saved on the stack is the address of the next instruction that would normally have been executed. All software can enable and disable some of the trap conditions with a single instruction.

**trap enables** Three bits in the Processor Status Word that control the processor’s action on certain arithmetic exceptions.

**two's complement** A binary representation for integers in which a negative number is one greater than the bit complement of the positive number.

**two-way associative cache** A cache organization which as two groups of directly mapped blocks. Each group contains several blocks for each index position in the cache. A block of data from main memory can go into either group at its proper index position. A two-way
Glossary

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 procedures calls using the current contents of the FP register and the FP register contents stored on the stack for each call frame.

**urgent interrupt** An interrupt received on interrupt priority levels 24 through 31. These can be generated only by the processor for the interval clock, serious errors, and power fail.

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

**user privileges** The privileges granted a user by the system manager. 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 register (R2, R3, ..., R11, AP, FP, SP, PC) are preserved across procedure calls.

**variable-length bit field** A set of zero of 32 contiguous bits located arbitrarily with respect to byte boundaries. A variable bit field is specified by four attributes: 1) the address A of a byte, 2) the bit position P of the starting location of the bit field with respect to bit 0 of the byte address A, 3) the size, in bits, of the bit field, and 4) whether the field is signed or unsigned.

**vector** 1. 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 a 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 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 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

Abort 296, 303
Absolute mode addressing 15, 111 to 113
Absolute indexed mode addressing 102
Absolute queue 176 to 179
Access modes 29, 30
Access control violation fault 302
Accumulator 79
Accuracy
  floating point instructions 125 to 128
Address 79
  manipulation instructions 20, 21, 171
  space
    physical 32 to 36, 43 to 47
    process 8, 43 to 45
    virtual 2, 7, 8, 28, 32 to 36, 43 to 47
Address manipulation instructions 20, 21
Address modes
  see also names of specific modes 83, 84
AP (Argument Pointer) 13, 48, 50
Architecture 1 to 3, 7, 8
  Control 1
  Extensibility 2
Argument 13
  lists 13, 351, 352
  procedures calls 351
  data types 359, 360
  descriptors 360 to 366
Argument Pointer (AP) 13, 48, 50
Arithmetic traps 14, 299 to 302
Array descriptor 363, 364
ASCII
  character set 331
  hexadecimal conversion 327 to 331
Assembler Radix notation 80
  Example 80
Autodecrement indexed mode addressing 102
Autodecrement mode addressing 92, 93
  definition 79
  Stack pointer 80
Autoincrement deferred indexed mode addressing 102
Autoincrement deferred mode addressing 15, 90, 91
Autoincrement indexed mode addressing 102
Autoincrement mode addressing 15, 88, 89
  definition 79
  stack pointer 80
Binary
  powers of 2 332
Bit-field
  data type 10, 72 to 75
Branch addressing 117, 118
Branch instructions 21 to 23, 205 to 212
Breakpoint fault 304, 305
Byte 59
Byte integer 57
Call frame 14
Calling sequence (procedure) 349 to 366
Carry condition code bit 53
Case instructions 23, 218, 219
Index

Character string
  data type  57, 62, 63
  instructions  19, 20, 233 to 248

COBOL output editing
  programming examples  383 to 386

Compatibility mode  25, 26, 317 to 326
  bit  55, 299
  exception  304
  instructions  318 to 320
  processor status word  318
  user environment  317

Conditions  366 to 377

Condition codes  53
  definition  14

Condition handling  367 to 377

Control instructions  205 to 230

Conversion
  ASCII — hexadecimal  331
  decimal — hexadecimal  327 to 329

Current access mode  298

Current mode field  55, 298

Cyclic redundancy check  245 to 248

D_floating format  9, 62

Data types  8 to 10
  representation  57 to 76
  tables  327 to 332

Decimal overflow trap enable bit  54, 297

Decimal string instructions  18, 19, 251 to 273

Decimal string overflow trap  301

Displacement deferred indexed mode addressing  102

Displacement deferred mode addressing  15, 99, 100

Displacement indexed mmode addressing  102

Displacement mode addressing  15, 97 to 99.

Divide by zero floating fault  302

Divide by zero trap 301
  floating or decimal string  301
  integer  300

Dynamic string descriptor  362

Edit instruction  19, 275 to 293

Edit pattern operators  282 to 293

Exception vector  36

Exceptions  25, 27, 295 to 315

Executive mode  55

F_floating format  9, 61

Failures
  System exceptions  308

Fault  296, 303

First Part Done flag  55, 298

Fixed format floating output
  programming example  382, 383

Floating overflow fault  302

Floating overflow trap  300

Floating point
  data types  9, 57, 61, 62
  instructions  17, 18, 121 to 163

Floating underflow fault  302

Floating underflow trap  301

Floating underflow trap enable bit  297

FORTRAN statement evaluation
  programming example  386

Frame Pointer (FP)  13, 25, 48, 50

Function value return
  procedure calls  353

General mode addressing  85 to 109

General purpose registers  10, 11, 47 to 50

General register manipulation
  instructions  21

Halt instruction  24, 314

Handlers
  condition  366 to 377

422
Index

Hardware memory management 32
Hardware process control block 39
Hexadecimal
  addition 328 to 330
  conversions 327 to 329, 331
  multiplication 328, 330
  powers of 16 332
Immediate indexed mode addressing 102
Immediate mode addressing 15, 111, 112
Index mode addressing 100 to 109
Index instruction 20, 301
Instruction exceptions 303, 304
Instruction format 15, 16, 80
Instruction set (native mode)
  address manipulation 20, 21
  branch and jump 21 to 23,
  character string 19, 20, 233 to 248
  compatibility mode 25, 26, 318 to 320
  edit 19, 275 to 293
  general register manipulation 21
  index instruction 20, 174, 175
  index of mnemonics 333 to 340
  index of opcodes 341 to 348
  integer and floating point 17, 18, 121 to 163
  loop control 214 to 217
  packed decimal 18, 19, 251 to 273
  procedure control 23, 24
  queue 20, 176 to 195
Special instructions 165 to 203
  subroutine control 23
  variable-length bit field 20, 196 to 203
Integer
  data types 8 to 10, 57, 59 to 61
  divide by zero trap 300
  instructions 17, 18, 121 to 163
  overflow trap 300
  overflow trap enable bit 297
Interrupt
  description 295
  vector 36, 37
Interrupt Priority Level (IPL) 38, 54, 295 to 298
Interrupt Stack flag 55, 298
Interrupt stack not valid halt 308
Jump instructions 21 to 23, 208, 220
Kernel mode 55
Kernel stack not valid abort 308
Label descriptor 365, 366
Label incarnation descriptor 366
Leading separate numeric string 67, 68
Literal mode addressing 93 to 97
Longword 10, 60
Longword integer 57
Loop
  control instructions 213 to 217
  programming example 387, 388
Machine check exception 308
Memory
  management 7, 32, 45, 46
  mapping registers 32 to 36
  physical
    see also Physical address
    space 32 to 36, 43 to 47
  virtual
    see also Virtual address
    space 2, 7, 8, 28, 32 to 36, 43 to 47
Mnemonics
  instruction set 333 to 340
Modes
  processor access 29, 30
  addressing 79 to 118
    see also names of specific modes
  Multiprogramming 26, 43
Native mode
  instruction set 16, 121 to 293, 311
to 314
    see also Instruction set
Negative condition code bit 53
Nibble 45
Index

Notational conventions 3
  extents 4
  must be zero (MBZ) 5
  operational notation 4
  ranges 4
  reserved 5
  undefined 4
  UNPREDICTABLE 4

Numeric string data types 57, 63 to 68

Opcode reserved to customers fault 304

Opcode reserved to DIGITAL fault 304

Opcodes 15, 16, 80 to 82
  Operand
    addressing mode 11
    specifiers 83
    types 82, 83

Operation code (opcode) 82

Overflow condition code bit 53.

Packed decimal
  data types 8 to 10, 57, 68 to 69
  instructions 18, 19, 251 to 273
  strings 251 to 253

Page table entry (PTE) 34, 46, 47
  modification flag 47
  page frame number (PFN) 47
  protection code 47
  valid code 47

Page tables 34 to 36

Paging 43

PC (Program Counter) 11, 15, 48, 296, 305

PDP-11 compatibility
  see also Compatibility
    mode 16, 25, 26, 299, 317 to 326
    program environment 317 to 321

Physical address 33

Physical address space 32 to 36, 43 to 47

Physical memory 32

Pointer
  register usage 11

Powers of 16 332

Powers of 2 332

Previous access mode field 54, 55, 298

Priority
  interrupt 36, 37
  levels 38

Privilege 55
  instructions 7, 31 to 32

Privileged process 30 to 32, 52

Procedure
  calling 349 to 366
  control instructions 23, 24, 222 to 230
  definition 12
  descriptor 364, 365
  incarnation descriptor 365

Process
  context 39
    definition 26
    priority dispatching 27, 28
    switching 26, 27, 31, 32
    definition 26, 43
    page table 34 to 36
    paging 43
    virtual address space 8, 43 to 45

Processor Status Longword (PSL) 28, 29, 52 to 55, 296 to 299
  MOVPSL instruction 169

Processor Status Word (PSW) 14, 15, 52 to 54, 296, 297, 318

Program counter (PC) 11, 15, 48, 296, 305

Program counter addressing modes 110 to 117

Programming examples 379 to 388

Protection 47

PSL (Processor Status Longword) 28, 29, 52 to 55, 296 to 299

424
PSW (Processor Status Word)  14, 15, 52 to 54, 296, 297, 318
PTE (Page Table Entry)  34, 46, 47
Quadword  60, 61
Quadword integer  57
Queue
  absolute  176 to 179
data types  57, 70 to 72
definition  70 to 72, 176
instructions  20, 180 to 183, 188 to 195
self-relative  184 to 187
Register deferred indexed mode  102
Register deferred mode addressing  86, 87, 88
Register mode addressing  85, 86
Registers
  general purpose  10, 11, 47 to 50
  manipulation instructions  20, 21, 165 to 172
  procedure calls  357, 358
  special usage  49, 50
  storing data  75, 76
Relative deferred mode  111, 115 to 117
Relative mode  111, 114, 115
Reserved addressing mode fault  303
Reserved operand exception  303, 304
Scalar descriptor  361, 362
SIN function
  programming example  381, 382
Sort algorithm
  example  379 to 381
SP (Stack Pointer)  12, 13, 48, 50, 309, 318
Special function instructions  24
SPT (system page table)  35, 36, 47
Stack Pointer (SP)  12, 13, 48, 50, 309, 318
Stacks  12 to 14, 44, 50 to 52, 295, 296, 309, 310, 318
String descriptor  361, 362
Subroutine
  definition  12
Subroutine control instructions  23
Subscript range trap  301
Supervisor mode  55
System failures  308
System space  35, 36, 43 to 47
System page table (SPT)  35, 36, 47
Trace Enable bit  53, 54, 297
Trace Pending bit  55, 298
Tracing  305 to 308
Trailing numeric string  64 to 66
Translation not valid fault  302
Trap  295
Trap-enable flags  53, 5451
Unconditional branch and jump instructions  21 to 23
User mode  55
Variable length bit field
  data types  57, 72 to 75
  Example  74, 75
  instructions  196 to 203
  programming example  386 to 388
Varying string descriptor  362
VAX-11
  architecture  1 to 3, 7, 8
  conditions  366 to 377
  instruction format  15, 16, 80 to 82
  instructions  121 to 293, 311 to 314
  see also Instruction set
Virtual address  33
Virtual address space  2, 7, 8, 28, 32
  to 36, 43 to 47
Virtual memory  4, 28, 34, 43, 20, 21
Word  59
Word integer  57
Index

Zero
  packed decimal representation  253

Zero condition code bit  53
VAX-11 ARCHITECTURE HANDBOOK
1979/80

Your comments and suggestions will help us in our continuous effort to improve the quality and usefulness of our handbooks.

What is your general reaction to this handbook? (format, accuracy, completeness, organization, etc.)

What features are most useful?

Does the publication satisfy your needs?

What errors have you found?

Additional comments

Name

Title

Company

Company Dept.

Address

City State Zip

(staple here)
DIGITAL EQUIPMENT CORPORATION, Corporate Headquarters: Maynard, Massachusetts 01754, Telephone (617) 897-5111—SALES AND SERVICE OFFICES; UNITED STATES—ALABAMA, Birmingham, Huntsville • ARIZONA, Phoenix, Tucson • CALIFORNIA, El Segundo, Oakland, Sacramento, San Diego, San Francisco, Santa Ana, Santa Barbara, Santa Clara • COLORADO, Colorado Springs, Denver • CONNECTICUT, Fairfield, Meriden • FLORIDA, Miami, Orlando, Tampa • GEORGIA, Atlanta • HAWAII, Honolulu • ILLINOIS, Chicago, Peoria, Rolling Meadows • INDIANA, Indianapolis • IOWA, Bettendorf • KENTUCKY, Louisville • LOUISIANA, New Orleans • MARYLAND, Baltimore, Odenton • MASSACHUSETTS, Boston, Springfield, Waltham • MICHIGAN, Detroit • MINNESOTA, Minneapolis • MISSOURI, Kansas City, St. Louis • NEBRASKA, Omaha • NEW HAMPSHIRE, Manchester • NEW JERSEY, Cherry Hill, Fairfield, Princeton, Somerset • NEW MEXICO, Albuquerque, Los Alamos • NEW YORK, Albany, Buffalo, Long Island, Manhattan, Rochester, Syracuse, Westchester • NORTH CAROLINA, Chapel Hill, Charlotte • OHIO, Cincinnati, Cleveland, Columbus, Dayton • OKLAHOMA, Tulsa • OREGON, Portland • PENNSYLVANIA, Allentown, Harrisburg, Philadelphia, Pittsburgh • RHODE ISLAND, Providence • SOUTH CAROLINA, Columbia • TENNESSEE, Knoxville, Nashville • TEXAS, Austin, Dallas, El Paso, Houston • UTAH, Salt Lake City • VERMONT, Burlington • VIRGINIA, Richmond • WASHINGTON, Seattle • WEST VIRGINIA, Charleston • WISCONSIN, Milwaukee • INTERNATIONAL ARGENTINA, Buenos Aires • AUSTRALIA, Adelaide, Brisbane, Canberra, Darwin, Hobart, Melbourne, Perth, Sydney, Tasmania, Townsville • AUSTRIA, Vienna • BELGIUM, Brussels • BOLIVIA, La Paz • BRAZIL, Rio de Janeiro, Sao Paulo • CANADA, Calgary, Edmonton, Halifax, Kanata, London, Montreal, Ottawa, Quebec City, Toronto, Vancouver, Winnipeg • CHILE, Santiago • DENMARK, Copenhagen • EGYPT, Cairo • FINLAND, Helsinki • FRANCE, Lyon, Marseille, Paris, Puteaux • HONG KONG • INDIA, Bombay • IRAN, Tehran • IRELAND, Dublin • ISRAEL, Tel Aviv • ITALY, Milan, Rome, Turin • JAPAN, Osaka, Tokyo • MEXICO, Mexico City • NETHERLANDS, Amsterdam, Hague, Utrecht • NEW ZEALAND, Auckland, Christchurch, Wellington • NORTHERN IRELAND, Belfast • NORWAY, Oslo • PERU, Lima • PUERTO RICO, San Juan • SAUDI ARABIA, Jeddah • SCOTLAND, Edinburgh • SINGAPORE • SOUTH KOREA, Seoul • SPAIN, Madrid • SWEDEN, Gothenburg, Stockholm • SWITZERLAND, Geneva, Zurich • TAIWAN, Taoyuan • UNITED KINGDOM, Birmingham, Bristol, Ealing, Epsom, Leeds, Leicester, London, Manchester, Reading, Welwyn • VENEZUELA, Caracas • WEST GERMANY, Berlin, Cologne, Frankfurt, Hamburg, Hannover, Munich, Nurnberg, Stuttgart • YUGOSLAVIA, Ljubljana •