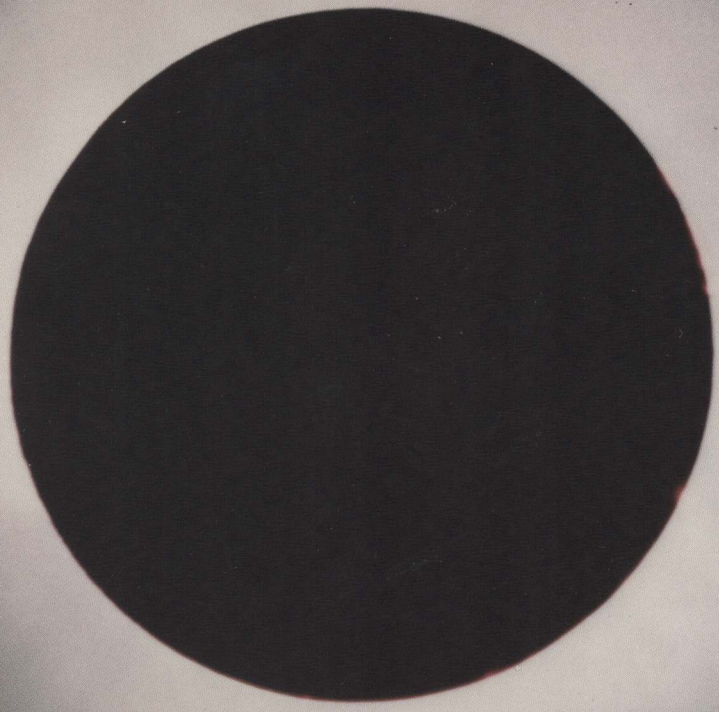
Data General

ECLIPSE® 32-Bit Systems
Principles of Operation



Programmer's Reference

DataGeneral

ECLIPSE® 32-Bit Systems
Principles of Operation

Programmer's Reference

Notice

Data General Corporation (DGC) has prepared this document for use by DGC personnel, customers, and prospective customers. The information contained herein shall not be reproduced in whole or in part without DGC's prior written approval.

DGC reserves the right to make changes in specifications and other information contained in this document without prior notice, and the reader should in all cases consult DGC to determine whether any such changes have been made.

THE TERMS AND CONDITIONS GOVERNING THE SALE OF DGC HARDWARE PRODUCTS AND THE LICENSING OF DGC SOFTWARE CONSIST SOLELY OF THOSE SET FORTH IN THE WRITTEN CONTRACTS BETWEEN DGC AND ITS CUSTOMERS. NO REPRESENTATION OR OTHER AFFIRMATION OF FACT CONTAINED IN THIS DOCUMENT INCLUDING BUT NOT LIMITED TO STATEMENTS REGARDING CAPACITY, RESPONSE-TIME PERFORMANCE, SUITABILITY FOR USE OR PERFORMANCE OF PRODUCTS DESCRIBED HEREIN SHALL BE DEEMED TO BE A WARRANTY BY DGC FOR ANY PURPOSE, OR GIVE RISE TO ANY LIABILITY OF DGC WHATSOEVER.

IN NO EVENT SHALL DGC BE LIABLE FOR ANY INCIDENTAL, INDIRECT, SPECIAL OR CONSEQUENTIAL DAMAGES WHATSOEVER (INCLUDING BUT NOT LIMITED TO LOST PROFITS) ARISING OUT OF OR RELATED TO THIS DOCUMENT OR THE INFORMATION CONTAINED IN IT, EVEN IF DGC HAS BEEN ADVISED, KNEW OR SHOULD HAVE KNOWN OF THE POSSIBILITY OF SUCH DAMAGES.

Ordering No. 014-000704

Rev. 03, August 1984

Revision History:

Original Release - December 1981
First Revision - March 1982
Second Revision - February 1983
Third Revision - August 1984

Preface

The *Principles of Operation 32-Bit* ECLIPSE® *Systems* manual explains the processor independent concepts, functions, and instruction set to an assembler programmer. The processor dependent information can be found in a companion manual — a processor specific functional characteristics manual.

The companion manual, which contains information such as physical memory size and instruction execution times, is organized with a structure similar to that of the *Principles of Operation 32-Bit* ECLIPSE® *Systems* manual. The similar structures make it easier to locate the cross-referenced information.

Another related manual, the ECLIPSE *MV/Family Instruction Reference Booklet*, presents a brief summary of the instruction set and related information. The reference booklet lists each instruction by assembler-recognizable mnemonic with a shorthand description of their function.

Organization

The 32-bit Principles of Operation manual contains 10 chapters.

Chapter 1 presents the system overview.

Chapters 2-9 present (in a functional framework) the processor independent concepts, functions, and instruction set. The chapters explain:

- Fixed-point computation
- Floating-point computation
- Stack management
- Program flow management
- Queue management
- Device management
- System and memory management
- ECLIPSE C/350 compatible instructions

Chapter 10 presents the instruction dictionary (alphabetical order).

Appendices A-E present anomalies, ASCII codes, powers of 2 table, fault codes, and the glossary.

Standard Symbols

The manual uses certain conventions and abbreviations.

[]	The square brackets indicate an optional argument. Omit the square brackets when you include an optional argument with an Assembler statement.
UPPERCASE and/or **Boldface**	Uppercase or boldface characters indicate a literal argument in an Assembler statement. When you include a literal argument with an Assembler statement, use the exact form.
lowercase and/or *Italic*	Lowercase or italic characters indicate a variable argument in an Assembler statement. When you include the argument with an Assembler statement, substitute a literal value for the variable argument.
*	An asterisk indicates multiplication. For instance, 2*3 means 2 multiplied by 3.
ac	The ac abbreviation indicates a fixed-point accumulator
acs	The acs abbreviation indicates a fixed-point accumulator called a *source accumulator.*
acd	The acd abbreviation indicates a fixed-point accumulator called a *destination accumulator.*
fac	The fac abbreviation indicates a floating-point accumulator
facs	The facs abbreviation indicates a floating-point accumulator called a *source accumulator.*
facd	The facd abbreviation indicates a floating-point accumulator called a *destination accumulator.*

Table of Contents

1 System Overview

Functional Capabilities 1-1
- Fixed-Point Computation 1-2
- Floating-Point Computation 1-3
- Stack Management 1-4
- Program Flow Management 1-5
- Queue Management 1-5
- Device Management 1-5
- System Management 1-6
- Memory Management 1-6
- ECLIPSE C/350 Compatible Instructions 1-8

Accessing Memory 1-8
- Current Segment 1-8
- Other Segments 1-9
- Memory Reference Instructions 1-9
- Address Modes 1-10
- Indirect and Effective Addresses 1-11
- Operand Access 1-12

Protection Capabilities 1-16
Summary 1-17

2 Fixed-Point Computing

Overview 2-1
Binary Operation 2-1
- Data Formats 2-1
- Move Instructions 2-3
- Arithmetic Instructions 2-3
- Carry Operations 2-5
- Shift Instructions 2-6
- Skip Instructions 2-7
- Overflow Fault 2-8
- Processor Status Register 2-9

Logical Operation 2-12
- Data Formats 2-12
- Logic Instructions 2-12
- Shift Instructions 2-14

Skip Instructions 2-14
Decimal and Byte Operations 2-15
Data Formats 2-16
Move Instructions 2-20
Arithmetic Instructions 2-22
Shift Instructions 2-22
Skip Instructions 2-22
Data Type Faults 2-23
Decimal Arithmetic Example 2-23

3 Floating-Point Computing

Overview 3-1
Data Formats 3-1
Move Instructions 3-3
Floating-Point Arithmetic Operations 3-4
Appending Guard Digits 3-4
Aligning the Mantissas 3-5
Calculating and Normalizing the Result 3-5
Truncating or Rounding the Result 3-5
Storing the Result 3-6
Arithmetic Instructions 3-6
Addition 3-6
Subtraction 3-7
Multiplication 3-7
Division 3-8
Skip Instructions 3-8
Faults and Status 3-9

4 Stack Management

Overview 4-1
Wide Stack Operations 4-1
Wide Stack Registers 4-2
Wide Stack Base 4-2
Wide Stack Limit 4-3
Wide Stack Pointer 4-3
Wide Frame Pointer 4-3
Wide Stack Register Instructions 4-4
Wide Stack Data Instructions 4-4
Initializing A Wide Stack 4-6
Wide Stack Faults 4-7

5 Program Flow Management

Overview 5-1
Program Flow 5-1
Related Instruction Groups 5-2
Execute Accumulator 5-2
Jump 5-2
Skip 5-2
Subroutine 5-4
Transferring Program Control to Another Segment 5-9
Subroutine Call 5-9

Subroutine Return 5-13
Fault Handling 5-13
Protection Violations 5-14
Umimplemented Instructions 5-19
Fixed-Point Overflow Fault 5-19
Floating-Point Overflow and Underflow Faults 5-20
Decimal and ASCII Data Faults 5-21
Stack Faults 5-25

6 Queue Management

Queues 6-1
Building a Queue 6-1
Queue Descriptor 6-2
Setting Up and Modifying a Queue 6-2
Examples 6-3
Queue Instructions 6-5

7 Device Management

Overview 7-1
Device Access 7-1
General I/O Instructions 7-3
Interrupts 7-5
Interrupt On Flag 7-6
Instruction Interruption 7-6
Interrupt Mask 7-6
Interrupt Servicing 7-6
Vectored Interrupt Processing 7-10
Base-Level Interrupt Processing 7-10
Intermediate-Level Interrupt Processing 7-11
Final Interrupt Processing 7-11

8 Memory and System Management

Overview 8-1
Segment Access and Address Translation 8-2
Segment Base Registers 8-2
Page Tables 8-4
Address Translation 8-6
Page Access 8-7
Central Processor Identification 8-11
Protection Violations 8-12

9 C/350 Programming

Overview 9-1
C/350 Registers 9-1
C/350 Stack 9-2
C/350 Faults and Interrupts 9-3
Expanding an ECLIPSE C/350 Program 9-3
Expanding an ECLIPSE C/350 Subroutine 9-4

C/350 Instructions 9-4
C/350 Memory Reference Instructions 9-4
Fixed-Point Instructions 9-7
Floating-Point Instructions 9-9
Program Flow Instructions 9-10
Stack Instructions 9-11

10 Instruction Dictionary

A Anomalies

C/350 Instruction Results A-1
Wide Instruction Opcodes A-1
Program Counter Wraparound A-1
Float/Fixed Conversions A-1
Address Wraparound A-2
C/350 Signed Divide Instructions A-2
NIO CPU Instructions A-2
Floating-Point Trap A-2
Floating-Point Numerical Algorithms A-2
C/350 Commercial Faults A-3

B ASCII Codes

C Powers of 2 Table

D Fault Codes

Protection Faults D-1
Stack Faults D-2
Decimal/ASCII Faults D-2

E Glossary

The Addressing Scheme E-1
Logical Addresses E-1
Segmentation E-1
Mapping and Demand Paging E-1
Page E-1
Page Table E-1
Protection E-2
The Instruction Set E-2
Wide Instructions E-2
C/350 Compatibility E-2
True and Impure Zero E-2
Normalized Format E-3
Magnitude E-3

Chapter 1
System Overview

Functional Capabilities

The ECLIPSE® 32-bit central processor -- hereafter called the processor -- provides facilities to manage data, to access memory, and to control program flow (see Figure 1.1).

The processor can perform fixed-point or floating-point computation, as well as stack, program, queue, device, system, and memory management. In addition, the processor contains the ECLIPSE C/350 compatible instructions for 16-bit program development and upward program compatibility.

This System Overview chapter provides a brief description of the processor functional capabilities, memory address space, and system protection capabilities.

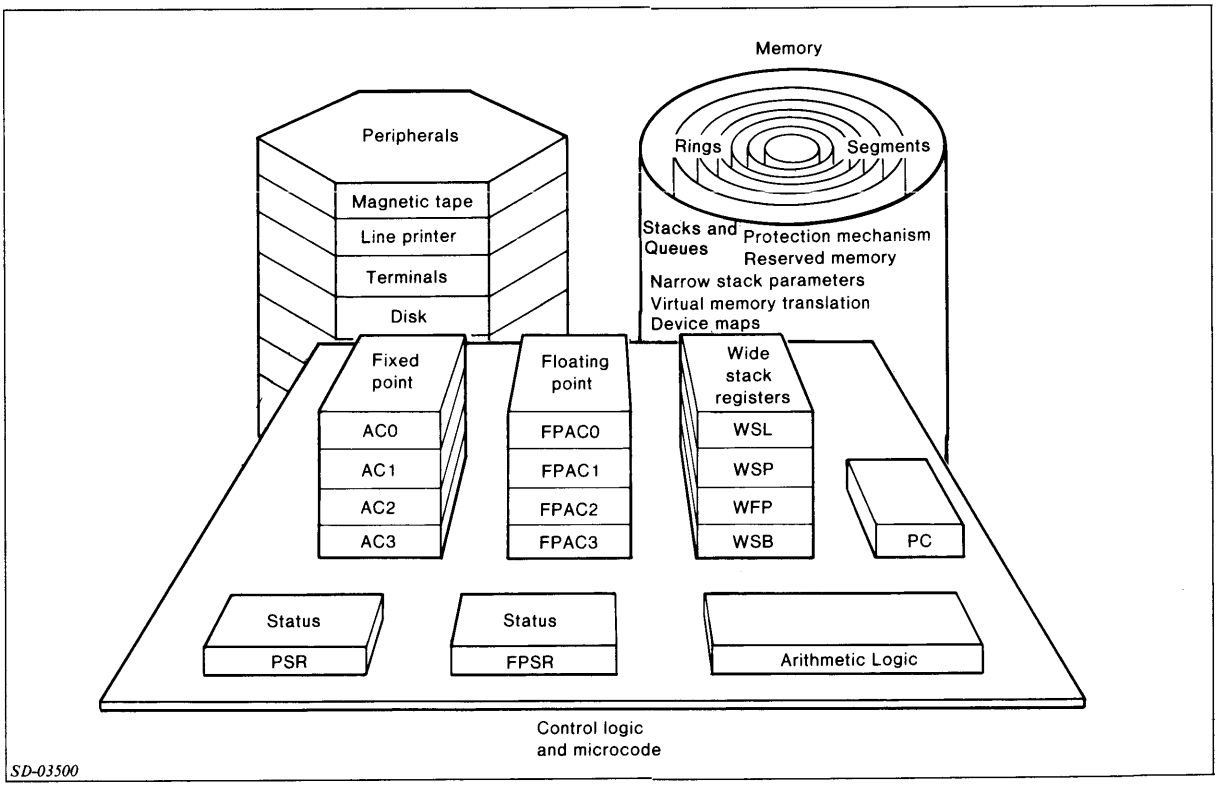


Figure 1.1 Functional components

Fixed-Point Computation

Fixed-point computation consists of fixed-point binary arithmetic with signed and unsigned 16-bit and 32-bit numbers. The processor also performs decimal arithmetic, logical operations, and manipulates 8-bit bytes.

The processor contains four 32-bit fixed-point accumulators (AC0-AC3) and a processor status register (PSR). The following two sections summarize the fixed-point registers. Refer to the Fixed-Point Computing chapter for additional information.

NOTE: *The lower numbered bit of a register (such as bit 0) is the most significant bit. The higher numbered bit (such as bit 31) is the least significant bit.*

Fixed-Point Accumulators

You access a fixed-point accumulator with instructions that manipulate a bit, byte, word, or double word (see Figure 1.2).

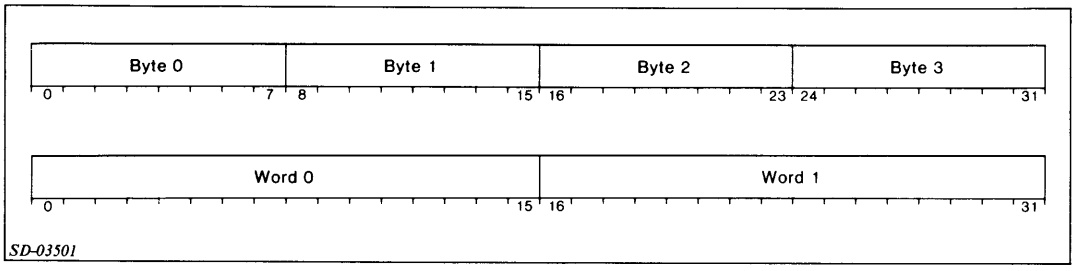


Figure 1.2 Fixed-point accumulator

A word or double word operand must begin on a word boundary (bit 0 or 16); a byte must begin on a byte boundary.

In addition to using an accumulator for fixed-point computation

- You can read a fault code in AC1, which the processor stores in the accumulator.
- You can load or build an instruction in an accumulator, and then execute it.
- You can use AC2 or AC3 in relative addressing (in place of the program counter).

Processor Status Register

The processor status register contains status flags such as an overflow fault service mask and a fixed-point overflow fault flag. The overflow fault service mask enables or disables the processor from servicing the fault. The processor sets the overflow fault flag when the results of a fixed-point computation exceed the processor storage capacity. The remaining flags are processor-dependent.

You can access the processor status register bits with instructions that set a bit or that test and skip on condition of a bit. Refer to the Fixed-Point Computing chapter for additional information.

Floating-Point Computation

Floating-point computation consists of floating-point binary arithmetic with signed, single precision (32-bits) and double precision (64-bits), numbers.

The processor contains four 64-bit floating-point accumulators (FPAC0- FPAC3) and a floating-point status register (FPSR). The following two sections summarize the floating-point registers. Refer to the Floating-Point Computing chapter for additional information.

Floating-Point Accumulators

You access a floating-point accumulator with instructions that manipulate single and double precision floating-point numbers (see Figure 1.3).

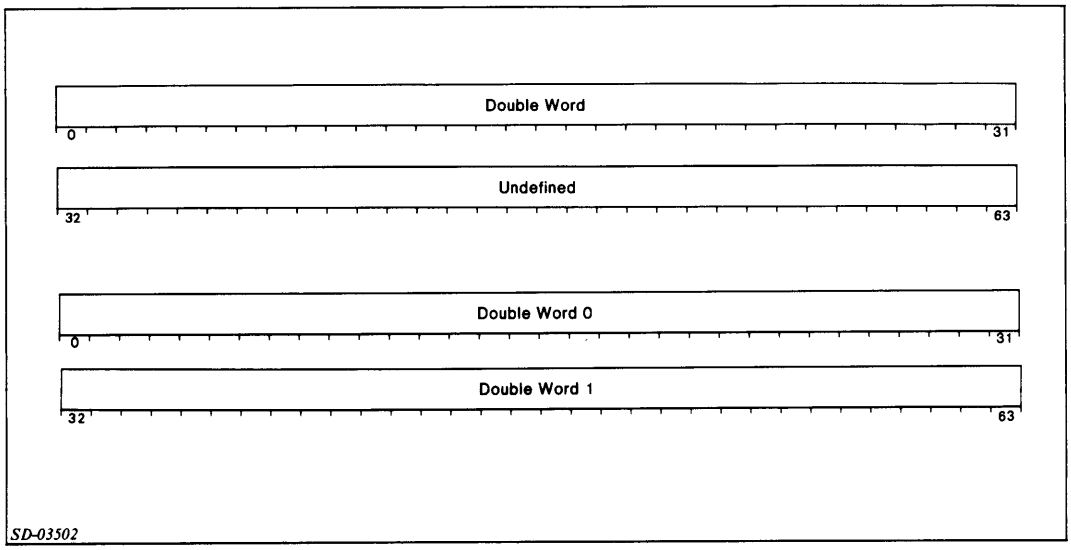


Figure 1.3 Floating-point accumulator

A single precision number requires a double word (two consecutive words), while a double precision number requires two double words (four consecutive words).

Floating-Point Status Register

The floating-point status register contains overflow and underflow fault flags, fault service mask, mantissa status flags, rounding flag, and processor status flags.

The processor sets an overflow or underflow fault flag when the result of a floating-point computation exceeds the processor storage capacity. The fault service mask enables or disables the processor from servicing a fault. The remaining flags provide processor status.

You can access the contents of the register with instructions to initialize it or to test and skip on a condition.

Stack Management

The processor contains facilities for narrow and wide stack management. A *stack* is a series of consecutive locations in memory. Typically, a program uses a stack to pass arguments between subroutine calls and to save the program state when servicing a fault. After executing a subroutine or fault handler, the processor restores the program and continues program execution.

The narrow stack consists of a contiguous set of words for supporting ECLIPSE C/350 program development and upward program compatibility. Narrow stack management includes three 16-bit narrow stack management parameters. Refer to the C/350 Programming chapter for additional information on the narrow stack.

The wide stack consists of a contiguous set of double words for supporting the 32-bit processor programs. Wide stack management includes four 32-bit wide stack management parameters, for each memory segment. (A memory *segment* is a logically addressable subset of memory. Refer to the Memory Management section for additional information on memory and segments.)

Wide stack management for the current segment also includes four 32-bit wide stack management registers. (The segment field of the program counter defines the *current segment*.) The Stack Management section summarizes the wide stack concepts. Refer to the Stack Management chapter for additional information on the wide stack.

The following list summarizes the wide stack management registers.

- The *wide stack base* (WSB) defines the lower limit of the wide stack.
- The *wide stack limit* (WSL) defines the upper limit of the wide stack.
- The *wide stack pointer* (WSP) addresses the current location on the wide stack.
- The *wide frame pointer* (WFP) defines a reference point.

You access a stack management register with instructions that load or store a register value.

The processor accesses the stack management registers to save or restore them when changing program flow between segments. Figure 1.4 shows the format of the registers.

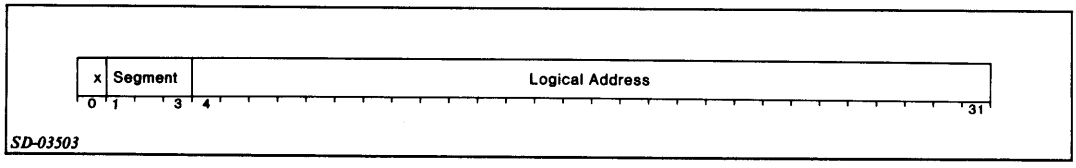


Figure 1.4 Wide stack management register format

where

x	Bit 0 is reserved for future use.
Segment	Bits 1-3 specify the segment location of the stack.
Logical Address	Bits 4-31 specify a logical address within the segment. Address wraparound can occur within the current segment.

Program Flow Management

Program flow management consists of controlling the program execution (such as calling a subroutine) and handling faults. The Program Flow Management section summarizes program control. Refer to the Program Flow Management chapter for additional information.

The processor controls program flow with a 31-bit program counter (PC). Figure 1.5 shows the format of the program counter.

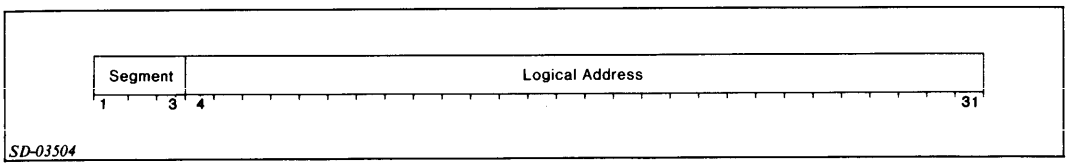


Figure 1.5 Program counter format

where

Segment — Bits 1-3 specify the current segment.

The processor provides specific procedures when modifying the current segment field.

Logical Address — Bits 4-31 specify a logical address within the segment.

During normal program flow, the processor increments bits 4-31 of the program counter. Thus, address wraparound occurs within the current segment.

Queue Management

Queue management consists of inserting, deleting, and searching for elements in a queue. A *queue* is a variable-length list of linked entries. Typically, an operating system uses queues to keep track of processes that it must run, such as printing files on a line printer.

Refer to the Queue Management chapter for further information on the queue facilities and management.

Device Management

Device management entails transferring data between memory and a device. The processor can transfer data (bytes, words, or blocks of words) with the programmed I/O, the data channel I/O (DCH), or the high speed burst multiplexor channel (BMC). The Device Management section summarizes the three transfer facilities.

Common to the three transfer facilities are the I/O instructions, mapped or unmapped memory addressing, and the interrupt system. Refer to the Device Management chapter for additional information on using the I/O instructions and the interrupt system.

Programmed I/O

With the programmed I/O facility, you transfer bytes or words between an accumulator and a device. You can use the programmed I/O facility to transfer data with a slow speed device, or to initialize a data channel or a burst multiplexor channel.

Data Channel I/O

With the data channel I/O, you initiate a transfer of words between memory and a device. The data channel accesses memory directly (with or without a device map). Thus, the data transfer bypasses the accumulators.

High Speed Burst Multiplexor Channel

With the burst multiplexor channel, you initiate a transfer of blocks of words between memory and a device. The burst multiplexor channel accesses memory directly (with or without a device map). Thus, the data transfer bypasses the accumulators.

System Management

System management provides facilities that determine processor dependent configurations, such as the processor identification and the size of the main memory.

Refer to the Memory and System Management chapter for additional information.

Memory Management

The processor uses a *virtual memory* of 4 Gbytes. Virtual memory consists of eight segments and rings, which facilitate memory management. A *segment* is an addressable unit of memory that contains programs and data. A *ring* is a collection of protection mechanisms, which safeguards the contents of a segment.

Since rings and segments are similar and inter-related, the manual uses the term segment to indicate either term or both terms. For instance, the manual refers to crossing segments; although gaining access to another segment also requires a ring crossing.

The processor addresses a segment through a 0-7 numbering system. Each segment contains 512 Mbytes. Figure 1.6 illustrates the concept of the segments, the contents of which are:

- Segment 0

  The processor executes privileged and nonprivileged instructions as the kernel operating system.

- Segments 1-7

  The processor executes nonprivileged instructions in segments 1-7. Refer to the appropriate operating system programmer's manual for the implementation-dependent usage of the segments.

Memory management entails allocating the virtual memory to various software functions, and then defining the memory access restrictions. The processor imposes a few restrictions when allocating the virtual memory, such as executing the kernel of an operating system in segment 0, and executing the system calls in or below the segment where you call them. Refer to the Accessing Memory section for more information on the memory access restrictions.

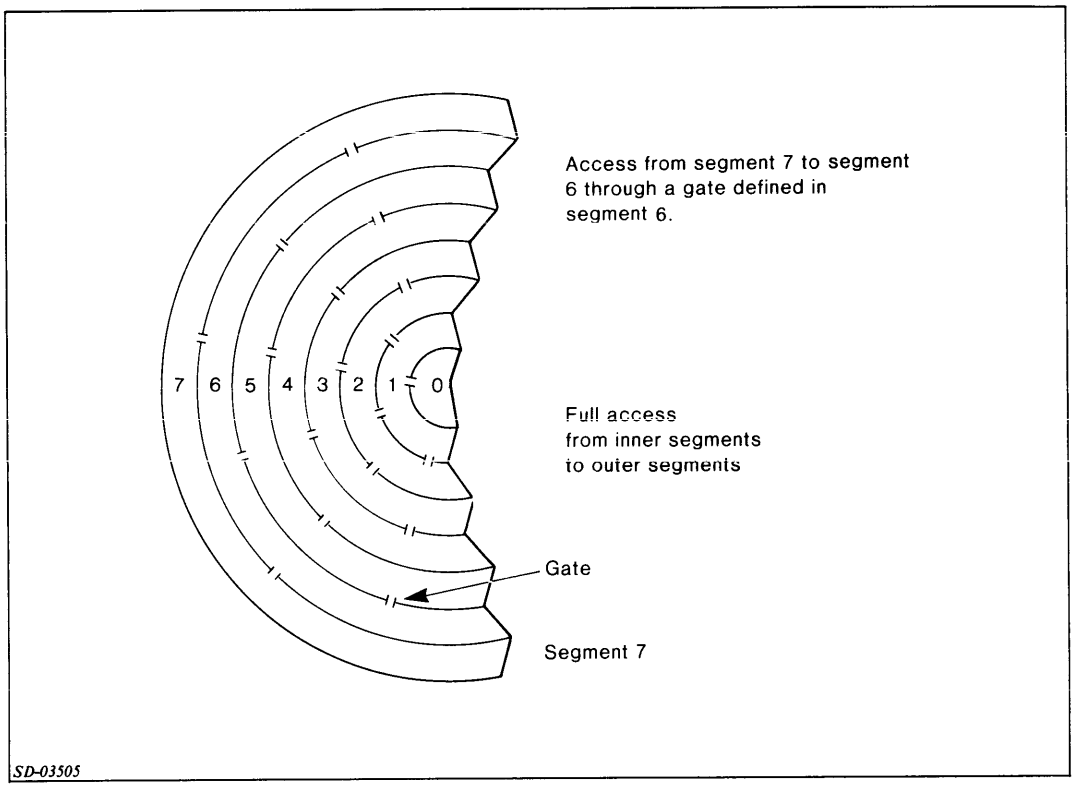


Figure 1.6 Virtual address space

Since the logical address space is larger than physical memory,

- The processor translates a logical address to a physical address.

  A logical address specifies a segment number and a logical location within the segment. You write programs using these logical addresses. The processor converts them to physical addresses, and then accesses the contents.

- The operating system stores portions of the virtual memory (pages of 2 Kbytes) on a disk.

The hardware facilities for address translation include eight memory management registers (SBR0- SBR7), which define eight memory segments and the access protocols. The processor performs an address translation as explained in the Accessing Memory section.

With a privileged instruction, you can access a memory management register to load or to store the contents of a register. Refer to the Memory and System Management chapter for additional information.

ECLIPSE C/350 Compatible Instructions

The processor contains an ECLIPSE C/350 compatible instruction set (and stack facilities) for 16-bit program development and upward program compatibility. Refer to the C/350 Programming chapter for additional information.

Accessing Memory

The processor addresses and accesses memory for an instruction or for an operand. To address memory, the processor uses a word as the standard unit of address. For instance, when loading a byte into an accumulator, the processor first resolves a word address, and then selects one of the two bytes.

The instruction that the processor accesses can be a word or a multiple of words. The operand can be a bit, byte, word, double word, or multiple of double words. You specify the address of the instruction or of the data with a memory reference instruction.

A *memory reference instruction* refers to a class of instructions that accesses memory for data or for another instruction. The memory reference instruction contains the information for

- Determining the effective address of an operand.

  The processor reads or writes an operand.

- Determining the effective address of the next nonsequential instruction.

  The processor modifies the program counter with the effective address, and then executes the instruction that the program counter identifies.

A memory reference instruction attempts to access memory in the current segment or in another segment. The validity of the access depends on a comparison of the access protocols permitted for the memory page and the type of access that the instruction attempts to perform. The access protocols are explained in the Current Segment and the Other Segments sections.

Current Segment

When a memory reference instruction addresses the current segment, the processor compares the page protocols with the type of access that the instruction requests, determining the validity of the reference. The *page protocols* are identified as a valid page, read access, write access, and execute access.

For instance, when loading a byte into an accumulator from the current segment, the processor reads the byte from memory if it resides where the page protocols permit a read access.

The processor also compares the segment field of every indirect address reference with the current segment. For accessing data (read or write access), indirect addressing can occur within the current segment or towards a higher numbered segment. For transferring program control (execute access), indirect addressing must occur in the current segment.

The processor aborts the access and services the protection violation fault for an invalid reference. Refer to the Memory and System Management chapter for further details on page accesses and the Program Flow Management chapter for more information on protection violation faults.

Other Segments

When executing a memory reference instruction that addresses another segment,

- The processor compares the current segment with the destination segment to determine the direction validity of the reference. The *destination segment* is the segment containing the operand or nonsequential instruction.

  A read or write access must be to the current or to a higher numbered segment. An execute access must be to the current or to a lower numbered segment.

- The processor compares the segment and page protocols with the type of access that the instruction requests to determine the access validity of the reference. The processor first checks the segment protocols, and then checks the page protocols.

  For a read or write access to a higher numbered segment, the segment protocol is a check for a valid segment. For an execute access to a lower numbered segment, the segment protocols are a check for a valid segment and gate. Refer to the Program Flow Management chapter for an explanation of a gate.

For instance, when loading a byte into an accumulator from a higher numbered segment, the processor reads the byte if it resides in a valid segment and page protocols permit a read access.

The processor aborts the access and services the protection violation fault for an invalid reference. Refer to the Memory and System Management chapter for further details on page accesses and the Program Flow Management chapter for more information on protection violation faults.

Memory Reference Instructions

Figure 1.7 shows the typical memory reference instruction formats for word addressing. Figure 1.8 shows the typical memory reference instruction formats for byte addressing. The instruction formats for word addressing contain an indirect (@) field. The instruction formats for word and byte addressing contain index and displacement fields, and also an optional accumulator (ac) field. The optional accumulator field specifies a source or destination accumulator in the range of zero to three.

For instance, with the ac field equal to zero (ac = 0) for a load accumulator instruction, the processor loads an operand from memory into the destination accumulator (AC0 or FPAC0).

> **NOTE:** *With the ac field, a fixed-point instruction specifies a fixed-point accumulator; a floating-point instruction specifies a floating-point accumulator.*

The combination of the index, displacement, and indirect (@) fields specify the effective address that contains the instruction or operand. To resolve the effective address, the processor first identifies the addressing mode, then any indirect address(es), and finally the effective address.

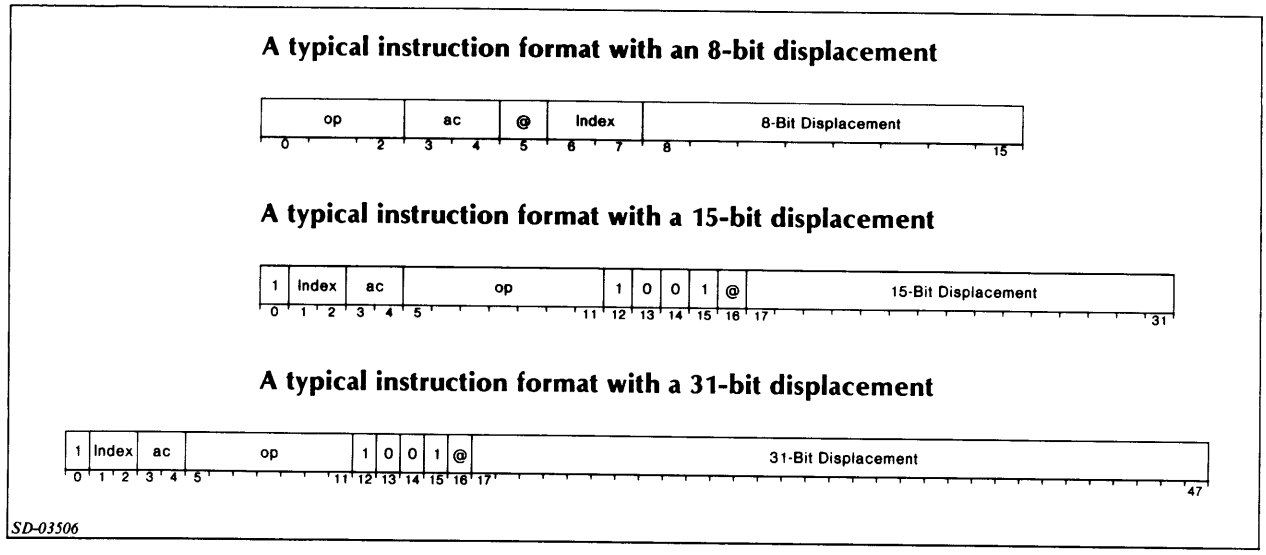


Figure 1.7 Memory reference instruction word addressing formats

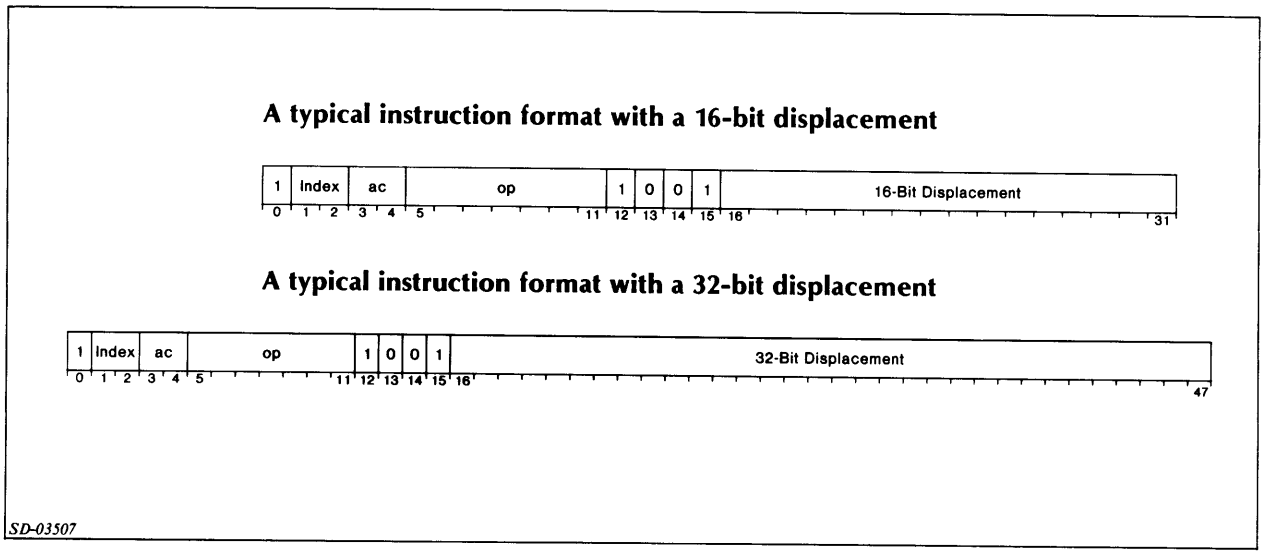


Figure 1.8 Memory reference instruction byte addressing formats

Address Modes

Using the index field (see Table 1.1), the processor determines if the instruction specifies an absolute or relative addressing mode. The Assembler (in conjunction with the appropriate pseudo-op) produces object code with absolute or relative addressing.

Absolute Addressing

For absolute addressing, the displacement field contains an indirect or an effective address. The address, expressed as an unsigned integer (8, 15, or 31 bits wide), specifies an addressing range as shown in Table 1.1.

With a few exceptions (LDA, LDB, LDI, LDIX, LEF, LSN, and XOP0), an assembler mnemonic of a memory reference instruction indicates the size and the range of the displacement. For instance, a memory reference instruction

- Without the X or L prefix, uses a standard displacement of 8 bits.
- With the X prefix, uses an extended displacement of 15 bits.
- With the L prefix, uses a long displacement of 31 bits.

NOTE: *When using an 8- or 15-bit displacement, the processor zero extends the displacement to 28 bits.*

Thus, the displacement becomes an indirect or an effective absolute address.

Relative Addressing

For relative addressing, the index field defines a register (see Table 1.1) the contents of which becomes a base address. The processor adds the base address to the displacement (8-, 15-, or 31-bit, two's complement integer). When using an 8- or 15-bit displacement, the processor sign extends the displacement to 31 bits.

In addition, if executing an instruction with an extended (15-bit) or long (31-bit) displacement, the processor adds a constant to the sum for program relative addressing. The additional increment adjusts the sum to address the first word of the displacement, which begins following the word that contains the instruction opcode. An instruction with an 8-bit displacement contains the displacement in the same word as the opcode.

Thus, the address becomes an indirect or effective relative address.

Indirect and Effective Addresses

When the indirect field equals zero, the absolute or relative address becomes the effective address. The processor translates an effective address to a physical address, and accesses the physical address.

When the indirect field equals one, the absolute or relative address becomes an indirect address (or pointer). The processor translates the indirect address to a physical address and uses the contents of that physical address as another indirect or direct address.

NOTE: *For a C/350 compatible instruction, the processor accesses a single word in memory as an indirect pointer; otherwise, the processor accesses a double word.*

The processor tests bit 0 of the pointer contents, which defines additional (if any) indirect addressing.

- When bit 0 equals zero, the contents become the effective address.

  The processor translates the effective address to a physical address and accesses it.

- When bit 0 equals one, the contents become another pointer.

  The processor continues to resolve pointers until bit 0 equals zero.

The processor can resolve up to 15_{10} pointers. However, for an instruction that can specify two indirect-addressing chains (such as WBLM), the total number of pointers for the two chains must be equal to or less than 15.

NOTE: *If the processor attempts to resolve more than 15 indirect addresses, a protection violation occurs.*

Address Mode	Index	Intermediate Logical Address*	Displacement Range	
			Prefix**	Octal Words (decimal)
Absolute	00	D		0 to 377 (0 to 255)
		D	X	0 to 077777 (0 to 32,767)
		D	L	0 to 17777777777 (0 to 2,147,483,647)
PC Relative	01	PC ± D		- 200 to + 177 (- 128 to + 127)
		PC + n ± D	X	- 40000 to + 37777 (- 16,384 to + 16,383)
		PC + n ± D	L	- 10000000000 to + 07777777777 (- 1,073,741,824 to + 1,073,741,823)
AC2 Relative	10	AC2 ± D		- 200 to + 177 (- 128 to + 127)
		AC2 ± D	X	- 40000 to + 37777 (- 16,384 to + 16,383)
		AC2 ± D	L	- 10000000000 to + 07777777777 (- 1,073,741,824 to + 1,073,741,823)
AC3 Relative	11	AC3 ± D		- 200 to + 177 (- 128 to + 127)
		AC3 ± D	X	- 40000 to + 37777 (- 16,384 to + 16,383)
		AC3 ± D	L	- 10000000000 to + 07777777777 (- 1,073,741,824 to + 1,073,741,823)

Table 1.1 Effective addressing

*The processor ignores bit 0 of PC, AC2, and AC3 when calculating the intermediate logical address.

The *n* variable in the PC relative addressing mode equals the number of words that precede the first word of the displacement, for the current instruction.

**The X or L corresponds to the prefix of an instruction mnemonic, which identifies the instruction as containing an extended (X) or long (L) displacement field.

Operand Access

Before accessing a memory operand (for fixed- or floating-point computation), the processor first resolves an effective address.

The processor accesses an operand as a bit, byte, several bytes, word, double word, or several double words. The following sections explain the word, byte, and bit accesses. (For the processor to access several bytes, it must first access a byte; to access several words or double words, it must first access a word.)

Word

The processor accesses a word operand for fixed-point computation. A fixed-point instruction mnemonic with a prefix of N (such as NADD) indicates a narrow or one word operand.

An instruction that requests a word (such as NLDA) supplies the effective address parameters to the processor. The processor then resolves the effective address.

Double Word

The processor accesses a double word operand for fixed-point or floating-point computation. A fixed-point instruction mnemonic with a prefix of W (such as WADD) indicates a wide or two word operand. A single precision floating-point instruction requires one double word, while a double precision instruction requires two double words.

An instruction that requests a double word (such as WLDA) supplies the effective address parameters to the processor. The processor then resolves the effective address, which points to the first word of the double word operand.

Byte

An instruction which requests a byte forms a byte pointer from the contents of an accumulator or from the contents of the index field and the 16- or 32-bit displacement. A byte pointer consists of an effective address and a byte indicator. The least significant bit of the byte pointer contains the byte indicator.

NOTE: *Byte addressing excludes indirect addressing.*

The processor identifies a byte as follows

- 16-Bit displacement

  For an instruction with a 16-bit displacement (such as XLDB), the processor extends the displacement to 29 bits (absolute addressing) or 32 bits (relative addressing), calculates the effective address, and then identifies the byte.

- 32-Bit displacement

  For an instruction with a 32-bit displacement (such as LLDB), the processor calculates the effective address, and then identifies the byte.

- Accumulator

  For an instruction that requires a byte pointer in an accumulator, you must first use a load effective byte address instruction (such as LLEFB). The load effective byte address instruction calculates an effective byte address, and then loads it into an accumulator.

Although identification of the bit numbers depend on the byte pointer location, the format of a byte pointer remains identical, regardless of the location. Figure 1.9 shows the formats for a byte pointer.

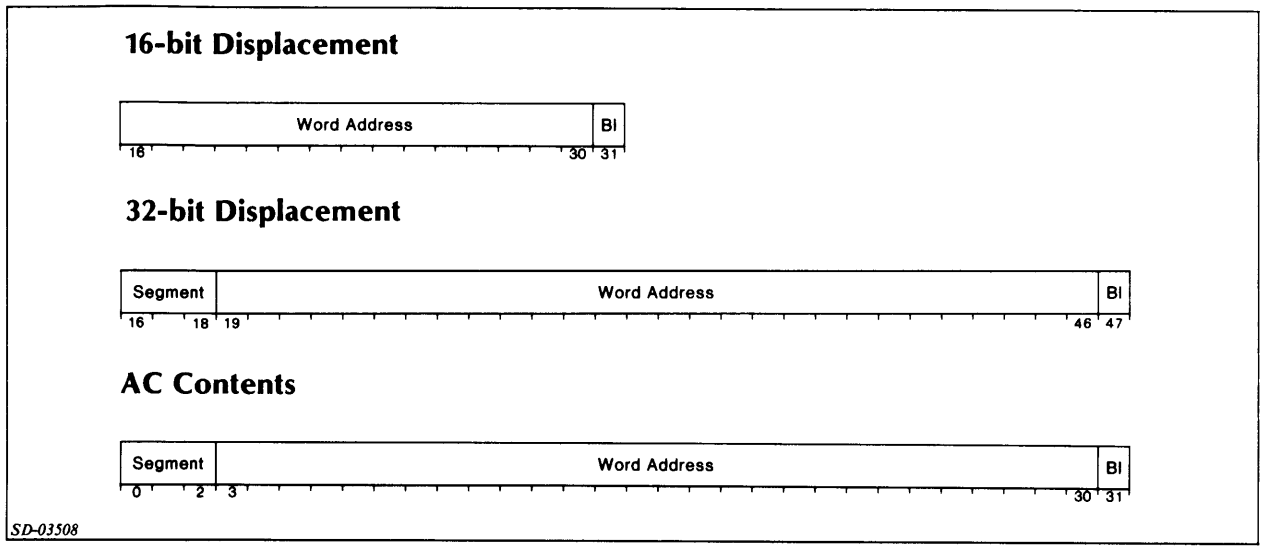


Figure 1.9 Byte pointer format

where

Segment	The segment field identifies the current or an outward memory segment.
Word Address	The word address field identifies a 16-bit word in the memory segment.
BI	The BI field identifies the byte. When BI field equals zero, the processor accesses the most significant byte (bits 0-7). When BI field equals one, the processor accesses the least significant byte (bits 8-15).

The processor accesses the word and then locates the byte (see Figure 1.10).

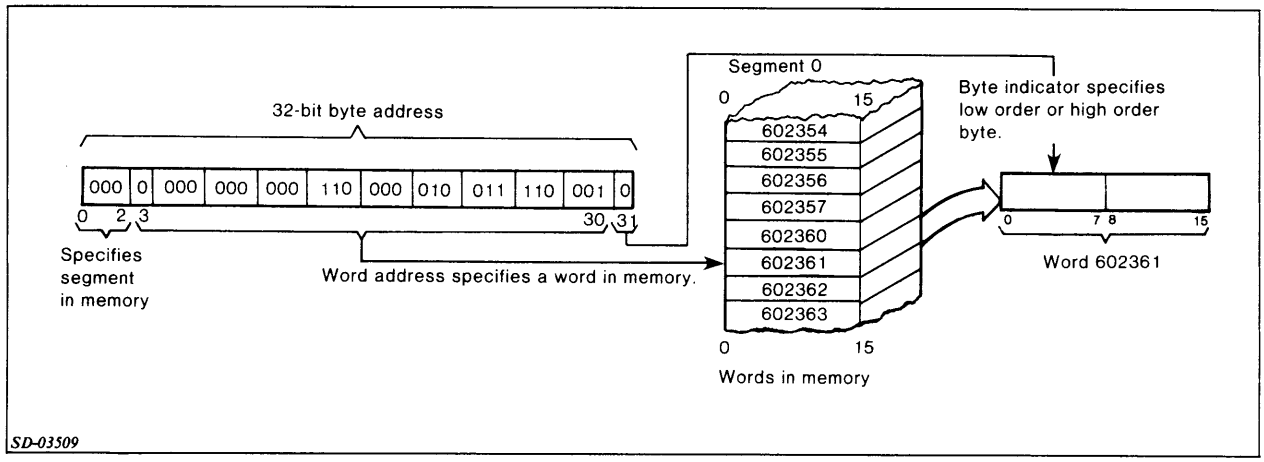


Figure 1.10 Byte addressing

Bit

An instruction that accesses a bit in memory (such as WBTO, WBTZ, WSNB, WSZB, and WSZBO) forms a bit pointer from the contents of two accumulators. The bit pointer is composed of a word pointer and a bit identifier. The word pointer consists of an effective address (in the ACS accumulator) and a word offset (in the ACD accumulator). The bit identifier is located in the least significant bits of the ACD accumulator.

Figure 1.11 shows the accumulator formats for the WBTO, WBTZ, WSNB, WSZB, and WSZBO instructions.

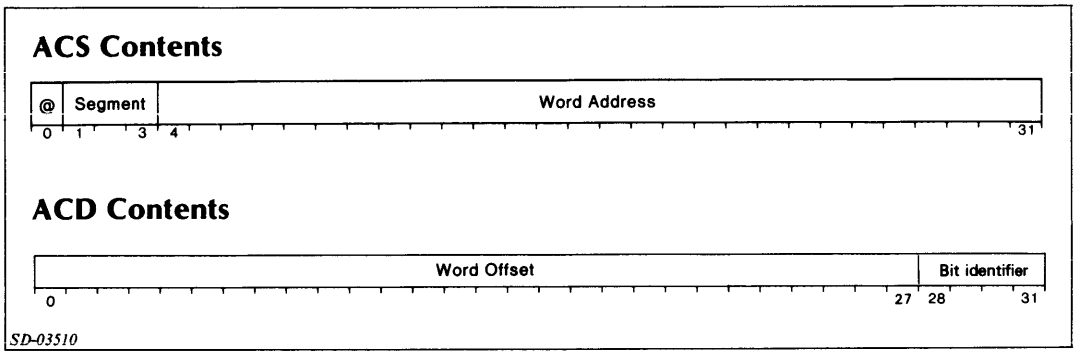


Figure 1.11 Bit pointer format

where

@	When the @ field equals one, it identifies an indirect address. When the @ field equals zero, it identifies a direct address.
Segment	The segment field identifies the current or an outward memory segment.
Word Address	The word address field identifies a 32-bit word in the memory segment.
Word Offset	The processor adds the word offset bits, an unsigned integer, to the effective address and arrives at a final word address (see Figure 1.12).
Bit Identifier	The bit identifier field specifies the bit position (0-15) in the final word.

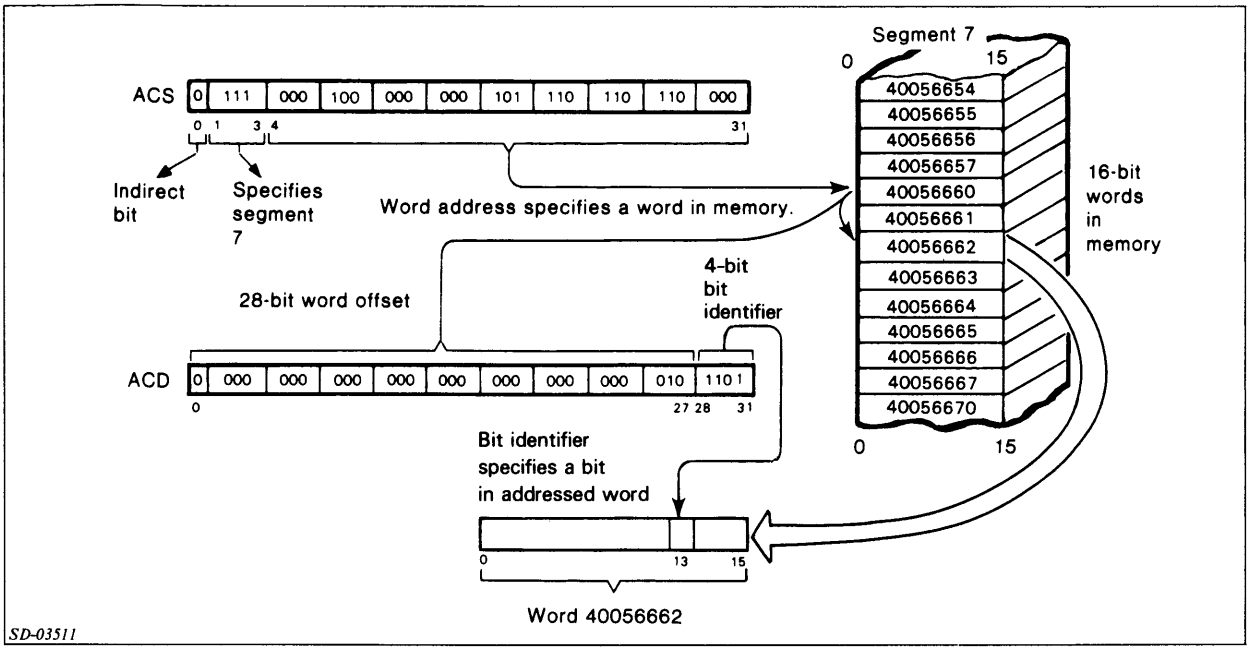


Figure 1.12 Bit addressing

The processor uses the ACS accumulator contents to calculate an effective address. If a bit instruction specifies the two accumulators as the same accumulator, then the effective address is zero in the current segment.

Protection Capabilities

While executing an instruction, the processor checks for the validity of a memory reference or an I/O operation (protection violation), a page reference (nonresident page), a stack operation, a computation, and a data format. Table 1.2 lists the validity checks (or faults).

Fault	Type
Nonresident page	Privileged
Protection violation	Nonprivileged
Stack operation	Nonprivileged
Fixed-point computation	Nonprivileged
Floating-point computation	Nonprivileged
Invalid decimal or ASCII data format	Nonprivileged

Table 1.2 Faults

If the processor detects an error, a nonprivileged or privileged fault occurs before executing the next instruction. A nonprivileged fault occurs when the processor detects a computation error. The processor limits the I/O and memory access using a hierarchical protection mechanism. For instance

- Before executing an I/O instruction, the processor checks the I/O validity flag in the current segment.
- Before executing a memory reference instruction, the processor checks the validity of the reference.

The processor executes an I/O or memory reference instruction when validity checks permit the access. Otherwise, the processor initiates a protection violation. Thus, an operating system can restrict access to the devices to specific segment(s).

Accessing and changing a protection mechanism requires a privilege instruction or data access, typically controlled by the operating system. Refer to the Program Flow Management chapter for further details on servicing a nonprivileged fault.

A privileged fault occurs when the processor detects a page fault, such as a nonresident memory page reference. Refer to the Memory and System Management chapter for further details on servicing a privileged fault.

Summary

The remainder of the *Principles of Operation 32-Bit ECLIPSE® Systems* manual explains the computation and management facilities. Chapters 2 through 9 present the facilities in a functional framework. Chapter 10 presents the instructions in alphabetical order, and the appendixes present C/350 anomalies and fault codes.

Chapter 2
Fixed-Point Computing

Overview

With fixed-point computations, the processor can add, subtract, multiply, and divide 16- and 32-bit signed (two's complement) and unsigned binary data. The processor also performs logical operations on 16- and 32-bit data.

In addition to the binary arithmetical and logical operations, the processor can manipulate 8-bit bytes (as alphanumeric ASCII data) and can perform binary coded decimal (BCD) arithmetic. The processor performs the byte manipulation with the fixed-point operations. The processor performs the BCD arithmetic with the fixed- and floating-point operations.

Following a computation, the processor can shift (arithmetically or logically) the contents of an accumulator, and can skip on a condition (the result of the computation and/or shift). Finally, the processor can store the result in an accumulator or memory.

The Fixed-Point Computing chapter explains the various computations (binary, logical, and decimal and byte) and the processor status register.

Binary Operation

The processor performs fixed-point binary arithmetic in the arithmetic logic unit. You control the processor and arithmetic logic unit operations with the move, arithmetic, shift, and skip instructions.

Data Formats

The fixed-point arithmetic instructions require two's complement binary numbers. For instance, the ADD instruction adds two 16-bit two's complement binary numbers. The 16- and 32-bit numbers must begin on word boundaries. Figure 2.1 shows the fixed-point accumulator formats for the 16- and 32-bit two's complement numbers.

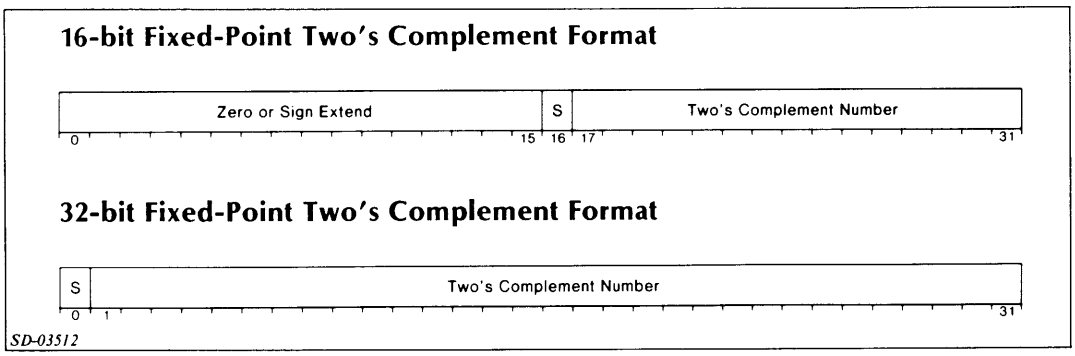


Figure 2.1 Fixed-point two's complement data formats

where

Zero or Sign Extend — The zero or sign extend bits contain 16 zeros or 16 ones.

For moving and computing narrow data, the processor sign extends narrow data when loading it into an accumulator. The processor sign extends narrow data before or after narrow data operations, when converting it to wide data.

S — The S bit equals the sign bit.

Bit 16 contains the sign bit for narrow data; bit 0 contains the sign bit for wide data. The sign bit equals zero for a positive number, and equals one for a negative number.

Two's Complement Number — The processor requires two's complement binary numbers for fixed-point arithmetic computation. Table 2.1 shows the precision of 16- and 32-bit two's complement binary numbers.

Form of Data	16-bit Precision	32-bit Precision
Unsigned	0 to 65,535	0 to 4,294,967,295
Signed	-32,768 to +32,767	-2,147,483,648 to +2,147,483,647

Table 2.1 Range of 16- and 32-bit two's complement numbers

Table 2.2 lists the instructions that explicitly convert 16-bit data to or from 32-bit data. Other sections list the instructions that convert the precision before or after another function. For instance, when loading narrow data (16-bit) from memory into an accumulator, the processor sign extends the number before loading it. When executing a narrow fixed-point instruction (NADD), the arithmetic logic unit sign extends the result.

Instruction	Operation
CVWN	Convert from 32-bit to 16-bit
SEX	Sign extend 16-bits to 32-bits
ZEX	Zero extend 16-bits to 32-bits

Table 2.2 Fixed-point precision conversion

Move Instructions

Table 2.3 lists the load and store accumulator instructions.

The wide block move instruction (WBLM) requires an effective address in an accumulator. Use a load effective address instruction (LLEF or XLEF) to calculate and to load the effective address into an accumulator.

Instruction	Operation
LDATS	Load accumulator with double word addressed by WSP
LNLDA	Narrow load accumulator
LNSTA	Narrow store accumulator
LWLDA	Wide load accumulator
LWSTA	Wide store accumulator
MOV *	Move and skip
NLDAI	Narrow load immediate
STATS	Store accumulator into double word addressed by WSP
WBLM	Wide block move
WLDAI	Wide load with wide immediate
WMOV	Wide move
WPOP	Wide pop accumulators
WPSH	Wide push accumulators
WXCH	Wide exchange accumulators
XCH *	Exchange accumulators
XNLDA	Narrow load accumulator
XNSTA	Narrow store accumulator
XWLDA	Wide load accumulator
XWSTA	Wide store accumulator

Table 2.3 Fixed-point data movement instructions

*ECLIPSE C/350 compatible instruction

Arithmetic Instructions

Tables 2.4 through 2.7 list the arithmetic instructions.

The ECLIPSE C/350 compatible instructions (such as, ADC, ADD, MUL, and DIVS) ignore bits 0-15 of the source accumulator. The results of C/350 compatible instructions generally leave bits 0-15 of the destination accumulator undefined, except where noted otherwise.

Instruction	Operation
ADC *	Add complement and skip
ADD *	Add and skip
ADDI *	Extended add immediate
ADI *	Add immediate
INC *	Increment and skip
LNADD	Narrow add memory word to accumulator
LNADI	Narrow add immediate
LWADD	Wide add memory word to accumulator
LWADI	Wide add immediate
NADD	Narrow add
NADDI	Narrow extended add immediate
NADI	Narrow add immediate
WADC	Wide add complement
WADD	Wide add
WADDI	Wide add with wide immediate
WADI	Wide add immediate
WINC	Wide increment (no skip)
WNADI	Wide add with narrow immediate
XNADD	Narrow add accumulator to memory word
XNADI	Narrow add immediate
XWADD	Wide add memory word to accumulator
XWADI	Wide add immediate

Table 2.4 Fixed-point addition instructions

*ECLIPSE C/350 compatible instruction

Instruction	Operation
LNSBI	Narrow subtract immediate
LNSUB	Narrow subtract memory word
LWSBI	Wide subtract immediate
LWSUB	Wide subtract memory word
NSBI	Narrow subtract immediate
NSUB	Narrow subtract
SBI *	Subtract immediate
SUB *	Subtract and skip
WSBI	Wide subtract immediate
WSUB	Wide subtract
XNSBI	Narrow subtract immediate
XNSUB	Narrow subtract memory word
XWSBI	Wide subtract immediate
XWSUB	Wide subtract memory word

Table 2.5 Fixed-point subtraction instructions

*ECLIPSE C/350 compatible instruction

Instruction	Operation
LNMUL	Wide multiply memory word
LWMUL	Wide multiply memory word
MUL *	Unsigned multiply
MULS *	Signed multiply
NMUL	Narrow sign extend multiply
WMUL	Wide multiply
WMULS	Wide signed multiply
XNMUL	Narrow multiply memory word
XWMUL	Wide multiply memory word

Table 2.6 Fixed-point multiplication instructions

*ECLIPSE C/350 compatible instruction

Instruction	Operation
DIV *	Unsigned divide
DIVS *	Signed divide
DIVX *	Sign extend and divide
HLV *	Halve (AC/2)
LNDIV	Narrow divide memory word
LWDIV	Wide divide memory word
NDIV	Narrow sign extend divide
WDIV	Wide divide
WDIVS	Wide signed divide
WHLV	Wide halve
XNDIV	Narrow divide memory word
XWDIV	Wide divide memory word

Table 2.7 Fixed-point division instructions

*ECLIPSE C/350 compatible instruction

Carry Operations

For fixed-point arithmetic operations, the processor maintains a carry flag (CARRY). The CARRY flag contains a value of zero or one. For instance, for an instruction that adds 16-bit data, the carry occurs from bit 16. For an instruction that adds 32-bit data, the carry occurs from bit 0.

You can initialize the value of the CARRY flag before a binary operation by executing an explicit carry instruction. Table 2.8 lists the instructions that initialize the CARRY flag. The processor retains the value of the CARRY flag for use with another instruction.

The processor changes the value of the CARRY flag as a result of executing an ECLIPSE MV/Family arithmetic instruction or an ECLIPSE C/350-compatible fixed-point instruction. For an ECLIPSE MV/Family arithmetic instruction, the processor loads the result of carry into the CARRY flag; it is not relative to its former value (as it is with an ECLIPSE C/350-compatible instruction). For an ECLIPSE C/350-compatible instruction, the processor complements the CARRY flag during

- addition when the two most significant bits (0 for unsigned and 1 for signed) and the CARRY flag produce a carry;
- subtraction when borrowing from the most significant bit.

With a C/350 compatible instruction, the processor initializes the CARRY flag, performs the binary operation on the data, and then modifies the CARRY flag (depending on the magnitude of the result).

Instruction	Operation
ADC *	Add complement with optional CARRY initialization
ADD *	Add with optional CARRY initialization
AND *	AND with optional CARRY initialization
COM *	One's complement with optional CARRY initialization
CRYTC	Complement CARRY
CRYTO	Set CARRY to one
CRYTZ	Set CARRY to zero
INC *	Increment with optional CARRY initialization
MOV *	Move with optional CARRY initialization
NEG *	Negate with optional CARRY initialization
SUB *	Subtract with optional CARRY initialization

Table 2.8 Initializing carry instructions

*ECLIPSE C/350 compatible instruction

Shift Instructions

The wide arithmetic shift instructions (WASH and WASHI) move 32 bits of an accumulator left or right (0 to 31 bit positions), depending on an 8-bit two's complement number. The 8 bits in the source accumulator for the WASH instruction or the 8 bits in the immediate displacement of the WASHI instruction contain the 8-bit number.

- With an 8-bit positive number, the processor shifts from 0 to 31 bit positions to the left, and zero extends the vacated bit positions. A fixed point overflow occurs if the sign bit changes.

  NOTE: *Shifting a negative number more than 31 bit positions to the left guarantees a fixed-point overflow.*

- With an 8-bit number equal to zero, no shifting occurs.
- With an 8-bit negative number, the processor shifts from 0 to 31 bits to the right, and sign extends the vacated bit positions. The processor drops the bits shifted from the least significant bit position.

  For instance, when the processor shifts +3 to the right one bit position, the result yields +1; shifting +1 to the right one bit position yields 0.

The ECLIPSE C/350 compatible arithmetic instructions (ADC, ADD, INC, MOV, and SUB) can shift an intermediate result one bit position or swap the two bytes (see Figure 2.2). The shift can be

- One bit to the left.

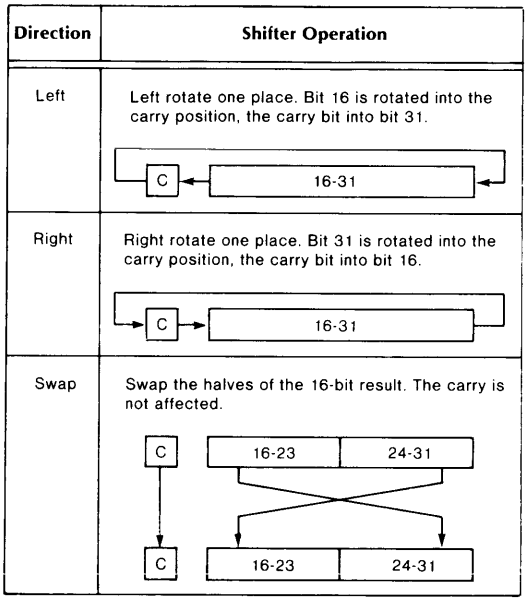
  The CARRY flag assumes the state of the most significant bit, and the least significant bit assumes the state of the CARRY flag.

- One bit to the right.

  The CARRY flag assumes the state of the least significant bit, and the most significant bit assumes the state of the CARRY flag.

- A swap of the most significant byte with the least significant byte.

  The processor preserves the state of the CARRY flag.



SD-03513

Figure 2.2 ECLIPSE C/350 compatible shift operations

Skip Instructions

With a skip instruction, the processor tests the result of an operation for a specific condition and directs the processor to skip the word or to execute the word after the skip instruction.

For an instruction that includes a skip option (such as ADD), the processor tests the result during its temporary storage. The processor can then save the result of the computation or ignore it. For an instruction that excludes a skip option (such as NADD), the processor stores the result in memory or an accumulator. You can then test the result with an explicit test and skip on condition instruction (such as skip on OVR reset -- SNOVR).

Tables 2.9 and 2.10 list the fixed-point skip on condition instructions. When a skip occurs, the processor increments the program counter by one and executes the second word after the skip instruction.

NOTE: *Be sure that a skip does not transfer control to the middle of a 32-bit or longer instruction.*

Instruction	Operation
ADC *	Add complement with optional skip
ADD *	Add with optional skip
INC *	Increment with optional skip
MOV *	Move with optional skip
NSALA	Narrow skip on all bits set in accumulator
NSALM	Narrow skip on all bits set in memory location
NSANA	Narrow skip on any bit set in accumulator
NSANM	Narrow skip on any bit set in memory location
SGE *	Skip if ACS greater than or equal to ACD
SGT *	Skip if ACS greater than ACD
SNOVR	Skip on OVR reset
SUB *	Subtract with optional skip
WCLM	Wide compare to limits and skip
WSALA	Wide skip on all bits set in accumulator
WSALM	Wide skip on all bits set in double word memory location
WSANA	Wide skip on any bit set in accumulator
WSANM	Wide skip on any bit set in double word memory location
WSEQ	Wide skip if ACS equal to ACD
WSEQI	Wide skip if equal to immediate
WSGE	Wide signed skip if ACS greater than or equal to ACD
WSGT	Wide signed skip if ACS greater than ACD
WSGTI	Wide skip if AC greater than immediate
WSKBO	Wide skip on AC bit set to one
WSKBZ	Wide skip on AC bit set to zero
WSLE	Wide signed skip if ACS less than or equal to ACD
WSLEI	Wide skip if AC less than or equal to immediate
WSLT	Wide signed skip if ACS less than ACD
WSNB	Wide skip on nonzero bit
WSNE	Wide skip if ACS not equal to ACD
WSNEI	Wide skip if AC not equal to immediate
WSZB	Wide skip on zero bit
WSZBO	Wide skip on zero bit and set bit to one
WUGTI	Wide unsigned skip if AC greater than immediate
WULEI	Wide unsigned skip if AC less than or equal to immediate
WUSGE	Wide unsigned skip if ACS greater than or equal to ACD
WUSGT	Wide unsigned skip if ACS greater than ACD

Table 2.9 Fixed-point skip on condition instructions

*ECLIPSE C/350 compatible instruction

Overflow Fault

The processor checks for a fixed-point overflow when attempting division by zero or when calculating a two's complement number. An overflow occurs if the result is too large to store in memory or in a fixed-point accumulator. At the end of the current instruction cycle, the processor sets the overflow flag (OVR) to one. The processor status register contains the OVR flag. Refer to the Program Flow Management chapter for information on fault handling.

Instruction	Operation
DSZTS	Decrement the double word addressed by WSP (skip if zero)
INC *	Increment and skip
ISZTS	Increment the double word addressed by WSP (skip if zero)
LNDSZ	Narrow decrement and skip if zero
LNISZ	Narrow increment and skip if zero
LWDSZ	Wide decrement and skip if zero
LWISZ	Wide increment and skip if zero
XNDSZ	Narrow decrement and skip if zero
XNISZ	Narrow increment and skip if zero
XWDSZ	Wide decrement and skip if zero
XWISZ	Wide increment and skip if zero

Table 2.10 Fixed-point increment or decrement word and skip instructions

*ECLIPSE C/350 compatible instruction

Processor Status Register

The processor contains a 16-bit processor status register (PSR), which retains information about the status of fixed-point computations. You access the register with instructions that test and set the register contents. Refer to the Skip section for a list of the instructions that test the register contents. Table 2.11 lists the instructions that set the register contents.

Instruction	Operation
BKPT	Breakpoint
FXTD	Disable fixed-point trap (resets OVK and disables trap)
FXTE	Enable fixed-point trap (sets OVK and enables trap)
LCALL	Call subroutine
LPSR	Load PSR into AC0
PBX	Pop block and execute
SPSR	Store PSR from AC0
WPOPB	Wide pop block
WRSTR	Wide restore
WDPOP	Wide pop context block
WRTN	Wide return
WSAVR	Wide save and set OVK to zero
WSAVS	Wide save and set OVK to one
WSSVR	Wide special save and set OVK to zero
WSSVS	Wide special save and set OVK to one
XCALL	Call subroutine
XVCT	I/O vector interrupt

Table 2.11 PSR manipulation instructions

*ECLIPSE C/350 compatible instruction

Figure 2.3 shows the format of the processor status register.

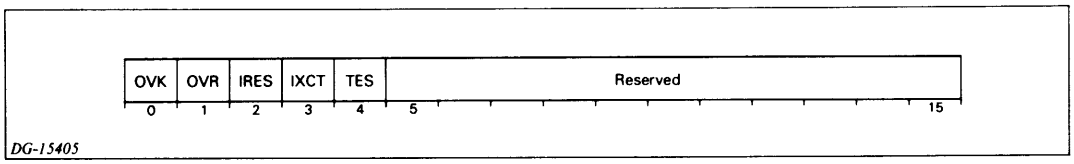


Figure 2.3 Processor status register format

NOTES: *The IRES, IXCT and TES bits are for hardware use. Do not modify the state of these bits; otherwise, results are unpredictable.*

Although all MV/Family machines implement Bits 0 and 1, some MV/Family machines may not implement the remaining PSR bits. Refer to the appropriate functional characteristics manual for more specific information.

where

OVK — The OVK bit is an overflow mask.

The processor (or you) enable fixed-point overflow detection and servicing by setting the OVK mask to one. You can set the OVK mask to one with the FXTE, LPSR, WSAVS, and WSSVS instructions. (See Table 2.11).

The processor saves or restores the status of the OVK mask when going to or returning from a subroutine or fault handler. For the processor to detect and service an overflow fault, the OVK mask must be set to one before the processor sets the OVR flag to one.

OVR — The OVR bit is an overflow flag.

The processor sets the OVR flag to one when it detects a fixed-point overflow condition.

The processor detects a fixed-point overflow condition when the result exceeds the 16-bit precision (for narrow data instruction) or 32-bit precision (for wide data instruction).

The overflow condition (*overflow*) exists for the duration of the fixed-point instruction that causes the overflow. The processor stores the transient *overflow* condition by performing a logical inclusive OR of *overflow* and the OVR flag before completing the instruction.

The OVR flag remains set to one until any of the following events occur

- I/O interrupt request

  Refer to the Device Management chapter for additional details.

- Fault detection and servicing

  Refer to the Program Flow chapter for additional details.

- Power up, I/O reset, or system reset

  Refer to the specific functional characteristics manual for additional details.

- Processor executes an instruction listed in Table 2.11

IRES

The IRES bit is an interrupt resume flag.

The processor sets the IRES flag when it interrupts a resumable instruction that requires the processor to save its state on the user stack. For example, when the processor interrupts a wide edit (WEDIT) instruction, the processor sets the IRES flag and saves the microstate on the user stack.

When a resumable instruction begins execution, it first tests the IRES flag. If the flag is 0, the instruction begins an initial execution. If the flag is 1, the instruction restores the state, resets the IRES flag to 0, and resumes execution.

NOTE: *Although the processor can interrupt some instructions, most instructions cannot be interrupted. Refer to the specific functional characteristics manual for additional information.*

IXCT

The IXCT bit is an interrupt-executed opcode flag.

When the processor executes the BKPT instruction, it pushes a wide return block onto the current stack. AC0 in the return block contains the one-word instruction (or the first word of a multi-word instruction). Then when returning program control, the PBX instruction (located at the end of the breakpoint handler) pops the wide return block and continues the normal program flow with the saved instruction in AC0.

If an interrupt occurs while executing the saved instruction (PC points to the BKPT instruction), the processor sets the IXCT flag in the PSR and pushes the opcode of the saved instruction onto the wide stack. Upon returning from the interrupt handler, the BKPT instruction tests the IXCT flag. If the flag is set, the BKPT instruction resets the flag to 0, pops the saved opcode of the interrupted instruction off the wide stack, and executes it.

TES

The TES bit contains the state of the Trap Enable (TE) flag of the Floating-Point Status Register (FPSR). The TES bit is applicable to systems with parallel floating-point units.

Before handling either an Interrupt or Page Fault, the processor must wait for any floating-point instruction executing in a parallel floating-point unit (FPU) to complete.

In order to guarantee that any floating-point fault is serviced in the proper context, the processor inhibits the floating-point trap until the completion of the Page Fault or the Interrupt service. To accomplish this, the processor sets bit 4 (TES) of the PSR to reflect the current value of the TE bit in the FPSR. The processor then clears the TE bit of FPSR to inhibit floating-point faults and services the page fault or interrupt.

Upon return from the service routine, the processor restores the FPSR TE bit from the PSR TES bit and clears the PSR TES bit. If the restored FPSR TE bit is 1, the processor services any pending floating-point traps after the next instruction boundary is crossed, such as after a WDPOP or WRSTR instruction.

Reserved

The processor sets the reserved bits to zero when storing them in memory. The processor ignores the reserved bits when loading the PSR.

NOTE: *Do not set the PSR bits 5 through 13 to store transient data while they are in memory (such as in a return block); these reserved bits must remain unused.*

When stored in memory, bits 14 and 15 are reserved for Data General software.

Logical Operation

The processor performs fixed-point logical arithmetic in the arithmetic logic unit. You control the processor and arithmetic logic unit operations with the move, logic, shift, and skip instructions.

With the AND, IOR, and XOR instructions, the processor performs the logical functions. The processor can then store the result into memory or it can test the result with a skip instruction, which either continues normal program flow or changes it.

Data Formats

The fixed-point logical instructions require the binary data to begin on word boundaries. For instance, an inclusive OR instruction (IOR) logically OR's two 16-bit binary values; a wide inclusive OR instruction (WIOR) logically OR's two 32-bit binary values. Figure 2.4 shows the 16- and 32-bit formats.

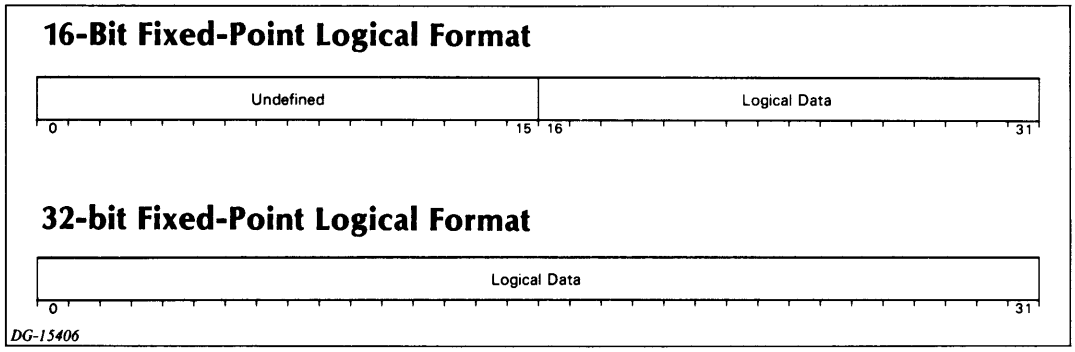


Figure 2.4 Fixed-point logical data formats

Logic Instructions

Table 2.12 lists the logical instructions. A wide set bit instruction (WBTO and WBTZ) requires an effective address in an accumulator. Use a load effective address instruction (LLEF or XLEF) to calculate and to load the effective address into an accumulator.

Instruction	Operation
ANC *	AND with complemented source
AND *	AND
ANDI *	AND immediate
COM *	Complement
IOR *	Inclusive OR
IORI *	Inclusive OR immediate
LOB *	Locate lead bit
LRB *	Locate and reset lead bit
NEG *	Negate
NNEG	Narrow negate
WANC	Wide AND with complemented source
WAND	Wide AND
WANDI	Wide AND immediate
WBTO	Wide set bit to one
WBTZ	Wide set bit to zero
WCOB	Wide count bits
WCOM	Wide complement (one's complement)
WIOR	Wide inclusive OR
WIORI	Wide inclusive OR immediate
WLOB	Wide locate lead bit
WLRB	Wide locate and reset lead bit
WLSN	Wide load sign
WNEG	Wide negate
WXOR	Wide exclusive OR
WXORI	Wide exclusive OR immediate
XOR *	Exclusive OR
XORI *	Exclusive OR immediate

Table 2.12 Logical Instructions

*ECLIPSE C/350 compatible instruction

Shift Instructions

Table 2.13 lists the logical shift instructions. With the ECLIPSE C/350 compatible shift instructions (AND, COM, and NEG), the processor can shift an intermediate result as explained for the ADC, ADD, and INC instructions. (See the Binary Operation section for further information).

Instruction	Operation
AND *	Logical AND with optional shift
COM *	Logical one's complement with optional shift
DLSH *	Double logical shift
LSH *	Logical shift
NEG *	Logical negate with optional shift
WLSH	Wide logical shift
WLSHI	Wide logical shift immediate
WLSI	Wide logical shift left immediate

Table 2.13 Logical shift instructions

*ECLIPSE C/350 compatible instruction

Skip Instructions

Table 2.14 lists the logical skip on condition instructions. When a skip occurs, the processor increments the program counter by one, and executes the second word after the skip instruction.

NOTE: *Be sure that a skip does not transfer control to the middle of a 32-bit or longer instruction.*

Instruction	Operation
AND *	AND with optional skip
COM *	One's complement with optional skip
NEG *	Negate with optional skip
WSNB	Wide skip on nonzero bit
WSZB	Wide skip on zero bit
WSZBO	Wide skip on zero bit and set bit to one

Table 2.14 Fixed-point logical skip instructions

*ECLIPSE C/350 compatible instruction

A wide skip on bit instruction (WSNB, WSZB, and WSZBO) requires an effective address in an accumulator. Use a load effective address instruction (LLEF or XLEF) to calculate and to load the effective address into an accumulator.

Decimal and Byte Operations

The processor performs decimal arithmetic (packed and unpacked) and 8-bit byte (or ASCII) manipulation. You control the various operations with the move, arithmetic, skip, and shift instructions. The move instructions include the instructions that convert, compare, and insert data.

The decimal arithmetic operations consist of

- Converting and moving decimal numbers between a floating-point accumulator and memory, and translating, scaling, and moving decimal strings between memory locations.

  The move instructions that convert one data type to another require an explicit data type description.

- Performing floating-point computations on the converted decimal numbers

  Refer to the Floating-Point Computing chapter for information on the floating-point arithmetic instructions.

The byte operations consist of

- Moving bytes from one memory location to another.
- Inserting or deleting bytes.

  You insert one or more bytes into a string first by moving the beginning part of the string to another location. Then, you move the bytes to be inserted to the other location, and finally you move the remainder of the string to the other location.

  You delete one or more bytes from a string first by moving the beginning part of the string to another location. Then, you skip the bytes to be deleted, and finally you move the remainder of the string to the other location.

- Converting from one data type to another data type.

  The move instructions that convert one data type to another require an explicit data type description.

- Comparing one data type to another data type or searching the string for a specific character.

  The skip instructions include the byte compare instructions even though they do not perform the skip function. A byte compare instruction stores the result of the comparison in an accumulator Use a skip on condition instruction to test the comparison.

Data Formats

The processor must know the format of the data before accessing it. Most instructions (such as fixed-point and floating-point instructions) imply a data format. However, for packed decimal (BCD) and unpacked decimal (ASCII) arithmetic with certain instructions (such as WEDIT, WLDI, WDMOV), the processor requires (in AC0 and/or AC1) an explicit data type indicator, as shown in Figure 2.5.

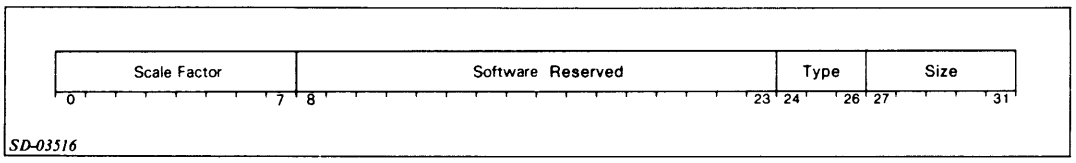


Figure 2.5 Explicit data type indicator

where

Reserved — The reserved field indicates that DGC reserves bits 8-23 for future use.

Scale Factor — The scale factor field indicates the scaling factor used by certain decimal instructions. Instructions which do not require a scaling factor ignore this field.

The scale factor (sf) is in the range $-128 < sf < 127$. For example, if the decimal string in memory is 932 then the following scale factors will represent the following numbers:

sf = -1 represents 9320
sf = 0 represents 932.
sf = 1 represents 93.2
sf = 5 represents .00932

Type — The type field identifies the type of data, as shown in Table 2.15.

Size — The size field indicates, as an unsigned integer, the length of the data. The following list explains the data type and corresponding size specification. Refer to Table 2.15 for examples.

Data Type	Size Field
0	Two less than the number of decimal digits and sign
1	Two less than the number of decimal digits and sign
2	One less than the number of decimal digits and sign
3	One less than the number of decimal digits and sign
4	One less than the number of decimal digits
5	One less than the number of decimal digits and sign With a data type 5, the processor expects an odd number for a size specification. If you specify an even size, the processor adds one to it (to make the size odd) and appends a zero digit to the most significant digit.
6	The number of bytes in the two's complement number You must specify a minimum of 1 byte
7	The number of bytes in the floating point number You must specify a minimum of 4 bytes

An unpacked decimal string contains one ASCII character in each byte (see Figure 2.6). Depending upon the data type and character location, the ASCII character represents a decimal digit, sign, or a digit and sign.

- Data types 0 and 1 combine the sign with a character.

  Refer to Table 2.16 for a list of the sign-positioned ASCII characters. Table 2.17 lists the nonsign-positioned ASCII characters.

- Data types 2 and 3 require the sign as a separate byte.

  The separate sign byte can be either the ASCII plus sign (+) -- 053_8 -- or the ASCII minus sign (-) -- 055_8. Table 2.17 lists the nonsign-positioned ASCII characters.

A packed decimal string contains two BCD digits per byte (see Figure 2.6). The most significant digit contains a zero if the decimal string contains an odd number of digits. The last byte must contain the least significant digit and the sign. The 15_8 (or D_{16}) represents the minus sign (-). The 14_8 or 17_8 (C_{16} or F_{16}) represent the plus sign (+).

Data Type	Meaning	Decimal Example	Characters in Each Byte Expressed in (Octal) or [Hex]	Data Type Indicator
0	*Unpacked decimal* - last byte combines the sign and the last digit	-397 +397	3 (063) 9 (071) P (120) 3 (063) 9 (071) G (107)	(000002)
1	*Unpacked decimal* - first byte combines the sign and the first digit	-397 +397	L (114) 9 (071) 7 (067) C (103) 9 (071) 7 (067)	(000042)
2	*Unpacked decimal* - last byte contains the sign	-397 +397	3 (063) 9 (071) 7 (067) - (055) 3 (063) 9 (071) 7 (067) + (070)	(000103)
3	*Unpacked decimal* - first byte contains the sign	-397 +397	-(055) 3 (063) 9 (071) 7 (067) + (070) 3 (063) 9 (071) 7 (067)	(000143)
4	*Unpacked decimal* - and unsigned	397	3 (063) 9 (071) 7 (067)	(000202)
5	*Packed decimal* - two BCD digits (or sign) per byte	-397 +397	39 (071) 7 - (175) 39 [39] 7 - [7D] 39 (071) 7+ (177) 39 [39] 7+ [7F]	(000243)
6	*Two's complement* - byte-aligned	-397 +397	(-615) = (176) (163) = (177163) (+615) = (001) (215) = (000615)	(000302)
7	*Floating point* - byte-aligned	-397 +397	(06771630000) [37] [E7] [30] [00] (06706150000) [37] [18] [D0] [00]	(000344)

Table 2.15 Explicit data types

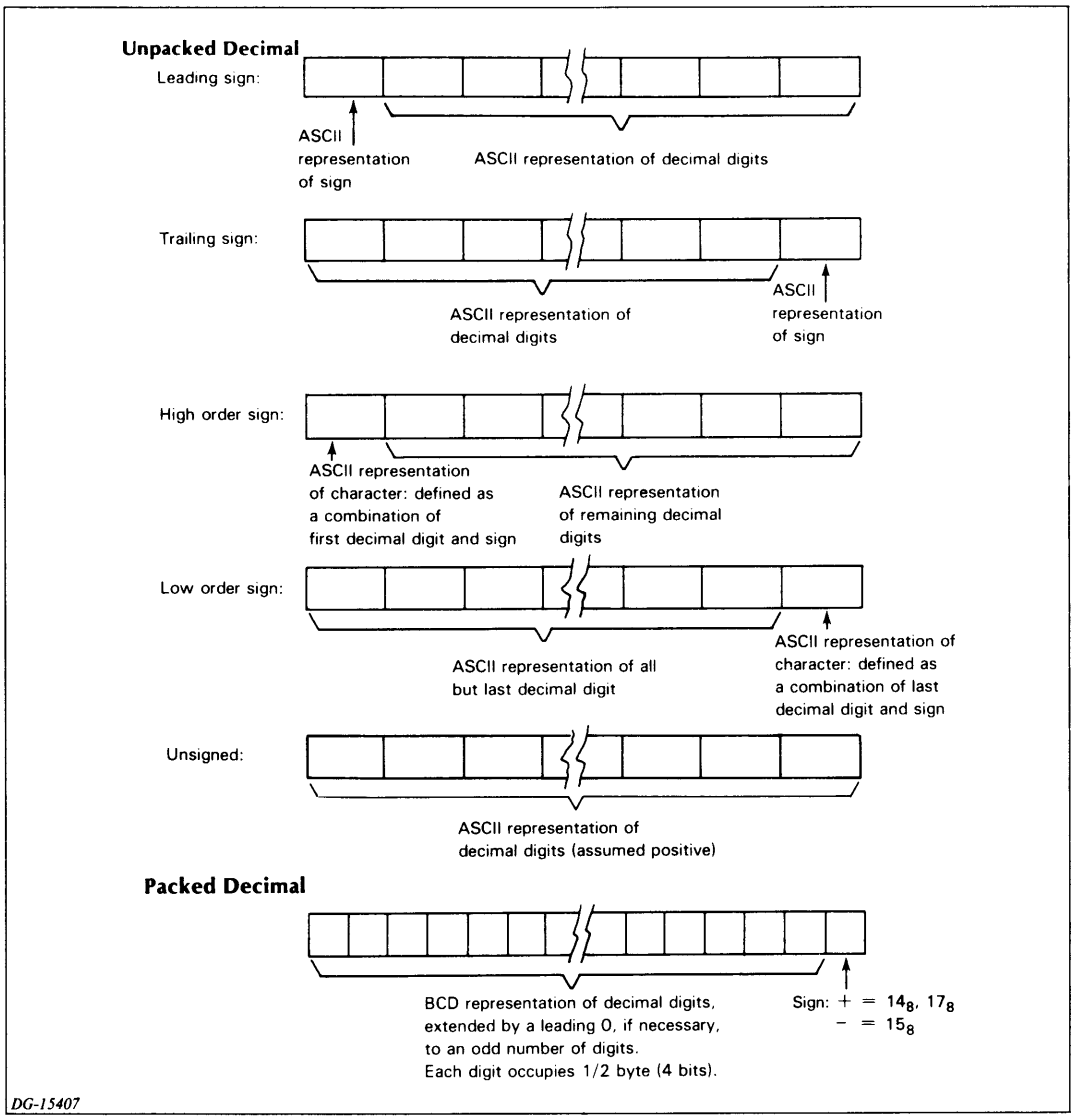


Figure 2.6 Packed and unpacked decimal data

Digit and Sign	ASCII Character (octal code)		Digit and Sign	ASCII Character (octal code)		Digit and Sign	ASCII Character (octal code)	
0+	space	(040)	5+	5	(065)	1-	J	(112)
0+	+	(053)	5+	E	(105)			
0+	{	(173)				2-	K	(113)
0+	0	(060)	6+	6	(066)			
			6+	F	(106)	3-	L	(114)
1+	1	(061)						
1+	A	(101)	7+	7	(067)	4-	M	(115)
			7+	G	(107)			
2+	2	(062)				5-	N	(116)
2+	B	(102)	8+	8	(070)			
			8+	H	(110)	6-	O	(117)
3+	3	(063)						
3+	C	(103)	9+	9	(071)	7-	P	(120)
			9+	I	(111)			
4+	4	(064)				8-	Q	(121)
4+	D	(104)	0-	-	(055)			
			0-	}	(175)	9-	R	(122)

Table 2.16 Sign and number combination for unpacked decimal

Digit	ASCII Character (octal code)
space	(040)
0	0 (060)
1	1 (061)
2	2 (062)
3	3 (063)
4	4 (064)
5	5 (065)
6	6 (066)
7	7 (067)
8	8 (070)
9	9 (071)

Table 2.17 Nonsign-positioned numbers for unpacked decimal

Move Instructions

The move instructions transfer formatted data between memory and a fixed-point accumulator or floating-point accumulator (FPAC) or between two memory locations. In addition to moving data, several instructions also convert, compare, or insert data.

Table 2.18 lists the instructions that move bytes of data. For an instruction that loads a byte into the least significant bits of a fixed-point accumulator, the processor zero extends the remaining bits. For an instruction that loads a byte into memory, the processor changes the addressed byte, while the other byte in the memory word remains intact.

Instruction	Operation
LLDB	Load byte
LSTB	Store byte
WCMT	Wide character move until true
WCMV	Wide character move
WCTR	Wide character translate and compare
WDMOV	Wide decimal move
WEDIT	Convert and insert string of decimal or ASCII characters
WLDB	Wide load byte
WSTB	Wide store byte
XLDB	Load byte
XSTB	Store byte

Table 2.18 Fixed-point byte movement instructions

The decimal move and convert instructions

- Convert packed decimal data to floating-point format when storing a decimal number in a floating-point accumulator.
- Convert floating-point data to packed decimal format when storing a decimal number in memory.

Table 2.19 lists the move and convert decimal/floating-point instructions.

Instruction	Operation
WLDI	Convert a decimal and load into FPAC
WLDIX	Convert a decimal, extend, and load it into four FPACs
WSTI	Convert FPAC data and load into memory
WSTIX	Convert the four FPACs and load into memory

Table 2.19 Fixed-point to floating-point conversion and store instructions

The move instructions require an effective word address and/or an effective byte address. Table 2.20 lists the instructions that calculate the address and store it in a fixed-point accumulator.

Instruction	Operation
LLEF	Load effective address
LLEFB	Load effective byte address
LPEF	Push address
LPEFB	Push byte address
WMOVR	Wide move right (convert byte pointer to word pointer)
XLEF	Load effective address
XLEFB	Load effective byte address
XPEF	Push effective address
XPEFB	Push effective byte address

Table 2.20 Load effective word and byte address instructions

The edit (WEDIT) instruction (with an edit subprogram) converts a decimal integer to a string of bytes, moves a string of bytes, or inserts additional bytes. Table 2.21 lists the edit subprogram instructions.

Instruction	Operation
DADI	Add signed integer to destination indicator
DAPS	Add signed integer to opcode pointer if sign flag is zero
DAPT	Add signed integer to opcode pointer if trigger is one
DAPU	Add signed integer to opcode pointer
DASI	Add signed integer to source indicator
DDTK	Decrement a word in the stack by one and jump if word is nonzero
DEND	End edit subprogram
DICI	Insert characters immediate
DIMC	Insert character j times
DINC	Insert character once
DINS	Insert character-a or character-b depending on sign flag
DINT	Insert character-a or character-b depending on trigger
DMVA	Move j alphabetical characters
DMVC	Move j characters
DMVF	Move j float
DMVN	Move j numerics
DMVO	Move digit with overpunch
DMVS	Move numeric with zero suppression
DNDF	End float
DSSO	Set sign flag to one
DSSZ	Set sign flag to zero
DSTK	Store in stack
DSTO	Set trigger to one
DSTZ	Set trigger to zero

Table 2.21 Edit subprogram instructions

Arithmetic Instructions

With the ECLIPSE C/350 compatible fixed-point add and subtract instructions, the processor computes the sum or difference of two unsigned BCD numbers in bits 28-31 of two accumulators. A carry, if any, is a decimal carry. With the wide decimal instructions, the processor adds one to or subtracts one from a decimal string or compares two decimal strings. Table 2.22 lists the arithmetic instructions.

Instruction	Operation
DAD *	Decimal add
DSB *	Decimal subtract
WDCMP	Wide decimal compare
WDDEC	Wide decimal decrement
WDINC	Wide decimal increment

Table 2.22 Arithmetic instructions

*ECLIPSE C/350 compatible instruction

Shift Instructions

With the ECLIPSE C/350 compatible hex shift instructions, the processor can move decimal results (in bits 16-31 of a fixed-point accumulator) either to the left or to the right. Table 2.23 list the hex shift instructions.

Instruction	Operation
DHXL *	Double hex shift left
DHXR *	Double hex shift right
HXL *	Hex shift left
HXR *	Hex shift right

Table 2.23 Hex shift instructions

*ECLIPSE C/350 compatible instruction

Skip Instructions

A load effective address instruction calculates a byte or word address that can point to a location outside the valid address space. When the processor executes a character manipulation instruction (such as WCMV) with an illegal address, a protection fault occurs. To avoid the protection fault, use

1. The load effective address instruction to calculate and load the byte or word address into an accumulator.
2. A skip on valid byte or word address instruction (VBP or VWP) to test the address.
3. The character manipulation instruction.

When a skip occurs, the processor increments the program counter by one and executes the second word after the skip instruction.

NOTE: *Be sure that a skip does not transfer control to the middle of a 32-bit or longer instruction.*

A skip instruction normally tests for a condition, and then modifies the program counter. However, the wide character compare (WCMP), the wide character translate and compare (WCTR), and the wide load sign (WLSN) instructions test for a condition, and then load a 0, -1, or +1 into AC1. You can then use a Wide Skip If Accumulator Equal instruction (WSEQ and WSEQI) to test the answer.

The wide character scan until true instruction (WCST) searches a string of bytes for one or more specified characters. When the instruction locates a byte, it stores the byte address in an accumulator.

Data Type Faults

The processor checks for a valid decimal or ASCII data type and for valid data when executing an instruction that requires an explicit data type description (such as WEDIT, WCTR, WSTI, or WCST). If either the data type or the data is invalid, the processor does not perform the instruction, but will service the fault before executing another instruction. Table 2.24 lists the decimal and ASCII fault codes. The first and second columns list the code that appears in AC1. The third column lists the type of return block pushed. The fourth column lists the instruction that caused the fault. The last column describes the conditions that can cause the fault.

Refer to the Program Flow Management chapter for more information on fault handling.

Decimal Arithmetic Example

Figure 2.7 illustrates an example of code written for execution under AOS/VS. The program

1. Accepts the decimal number from a terminal (in ASCII format).
2. Converts it to single precision floating-point format.
3. Performs the floating-point addition.
4. Converts the sum to ASCII format.
5. Displays it on the terminal.

Code Returned in AC1 Narrow	Code Returned in AC1 Wide	Return Block Type	Faulting Instruction	Meaning
000000	100000	2	EDIT, WEDIT	An invalid digit or alphabetic character encountered during execution of one of the following subopcodes: DMVA, DMVF, DMVN, DMVO, DMVS
000001	100001	1	LDIX, STIX	Invalid data type (7)
		3	EDIT, WEDIT WLDIX, WSTIX, WDMOV, WDCMP, WDINC, WDDEC	Invalid data type (6 or 7)
000002	100002	2	EDIT, WEDIT	DMVA or DMVC subopcode with source data type 5; AC2 contains the data size and precision
000003	100003	2	EDIT, WEDIT	An invalid opcode; AC2 contains the data size and precision
000004	100004	1	STI, LDI, WSTI, WLDI	Number too large to convert to specified data type . number $> (10^{16}) - 1$
			STIX, LDIX, WSTIX, WLDIX	Number too large to convert to specified data type. Number $> (10^{32}) - 1$
000006	100006	1	WLSN, WLDI, LSN, LDI LDIX, WLDIX	Sign code is invalid for this data type
		3	EDIT, WEDIT WDMOV, WDCMP, WDINC, WDDEC	
000007	100007	1	WLSN, WLDI, WLDIX, LSN LDI, LDIX	Invalid digit
		3	WDMOV*, WDCMP*, WDINC*, WDDEC*	

Table 2.24 Decimal and ASCII fault codes

*Only the required digits will be scanned for digit validity. Refer to the Functional Characteristics manual for your machine's requirement.

```
                .TITL   DECIMAL
                .ENT    START
                .NREL

                ;CONSTANTS
                .ENABLE SWORD
CON:    .TXT    "@CONSOLE"          ;GENERIC CONSOLE NAME
FCON:   202                         ;TYPE 4 AND 3 DECIMAL DIGITS
IBUF:   .BLK 5                      ;RESERVE 5 WORDS FOR NUMBER BUFFER

                ;PARAMETER PACKETS

;READ CONSOLE PACKET TO OPEN, READ, & WRITE

CONSOLE:  .BLK  22
          .LOC  CONSOLE+?ISTI
          ?RTDS+OFIO                ;DATA SENSITIVE I/O
          .LOC  CONSOLE+?ISTO
          0
          .LOC  CONSOLE+?IMRS
          -1
          .LOC  CONSOLE+?IBAD       ;?IBAD CONTAINS BYTE POINTER TO DATA PACKET

          .LOC  CONSOLE+?IRCL
          -1
          180.
          .LOC  CONSOLE+?IRNW
          0

          .ENABLE DWORD

          .LOC  CONSOLE+?IDEL
          -1
          .LOC  CONSOLE+?ETSP
          0
          .LOC  CONSOLE+?ETFT
          0
          .LOC  CONSOLE+?ETLT
          0
;END OF CONSOLE PACKET

START:    ?OPEN   CONSOLE           ;OPEN CONSOLE TO READ AND WRITE
            .
            .
            .
          ?READ   CONSOLE           ;ACCEPT A NUMBER FROM THE KEYBOARD
            .
            .
            .
          XNLDA   1,FCON            ;INIT FOR DATA TYPE 4
          XNLDA   3,CONSOLE+?IBAD   ;GET BYTEPTR FROM CONSOLE PKT
          WLDI    0
          FAS     0,0               ;SINGLE PRECISION FLOATING-POINT ADD
          XNLDA   1,FCON
          XNLDA   3,CONSOLE+?IBAD
AGAIN:    WSTI    0
          INC     1,1,SZC           ;INC BYTE COUNT AND SKIP IF WSTI TRUNCATES
          WBR     AGAIN             ;REPEAT WSTI

          ?WRITE  CONSOLE           ;DISPLAY THE SUM ON THE CONSOLE
            .
            .
            .
          ?CLOSE  CONSOLE           ;CLOSE THE CONSOLE
            .
            .
            .
          ?RETURN                   ;RETURN TO CLI
            .
            .
            .
          .END    START
```

SD-03518

Figure 2.7 Decimal arithmetic example

Chapter 3
Floating-Point Computing

Overview

With floating-point computations, the processor can add, subtract, multiply, and divide 32-bit (single precision) and 64-bit (double precision) sign magnitude data.

With the optional Intrinsic Instruction Set (IIS), the processor can perform trigonometric and logarithmic functions, exponentiation, and square root evaluation on 32-bit and 64-bit data.

Following a computation, the processor can convert a double precision value to a single precision value, or it can convert a single precision value to a fixed-point or decimal value. Then, the processor can test and skip on a condition that results from the computation or conversion. Finally, the processor can store the result in an accumulator or memory.

The Floating-Point Computing chapter explains the various computations (move, arithmetic, and skip), the Intrinsic Instruction Set, and the floating-point status register (FPSR).

Data Formats

The floating-point arithmetic and intrinsic instructions require normalized, sign magnitude numbers. You can use the floating-point normalize (FNOM) instruction to normalize raw floating-point data, which may or may not be normalized.

In addition, if a mantissa equals zero, the processor expects it to be a true zero. A *true zero* exists when the sign bit, exponent, and mantissa equal zero (all bits equal zero).

The single and double precision numbers must begin on word boundaries and must be within the value range of $5.4(10^{-78})$ to $7.2(10^{75}$. Figure 3.1 shows the floating-point formats.

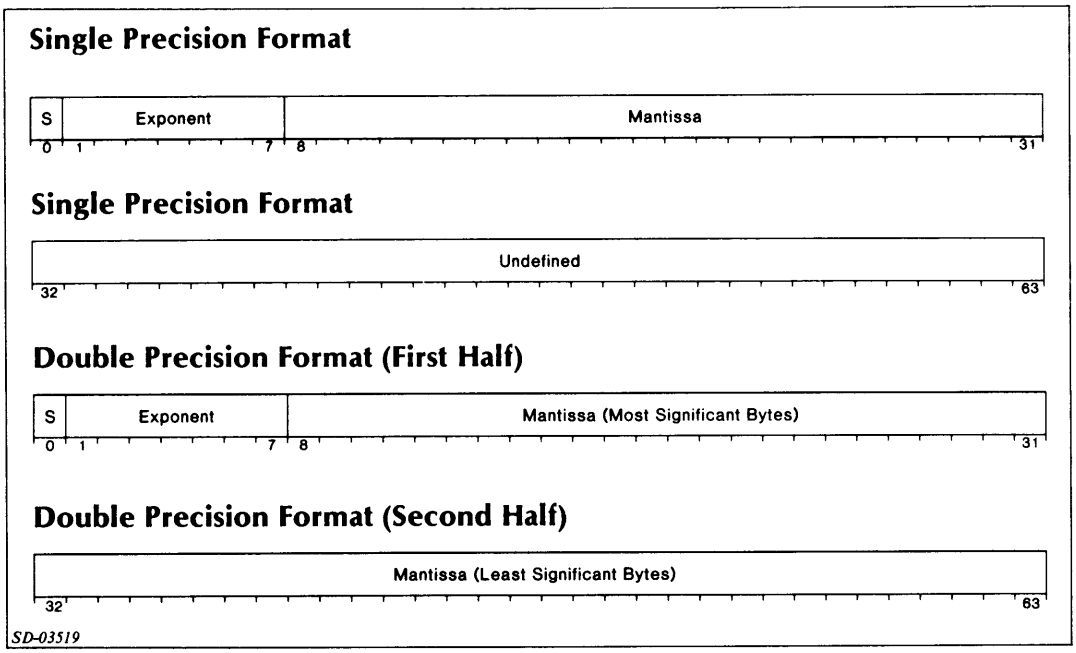


Figure 3.1 Floating-point data formats

where

S — The S bit equals the sign bit of the mantissa. The sign bit equals a zero for a positive number, and equals a one for a negative number.

Exponent — The exponent, expressed as an unsigned integer, equals 64_{10} greater than the true value of the exponent (excess 64 representation). The following exponents illustrate excess 64 representation numbers.

Exponent	True Value of Exponent
0	-64_{10}
64_{10}	0
127_{10}	$+63_{10}$

Mantissa — The mantissa, expressed as a fraction, implies that the location of the binary point is between bits 7 and 8.

The range of the mantissa for single precision is

$1/16$ to $1-2^{-24}$

and for double precision is

$1/16$ to $1-2^{-56}$

Table 3.1 lists the instructions that convert and move data between fixed-point and floating-point accumulators, convert a mixed number to a fraction, and scale a floating-point number.

Instruction	Operation
FEXP *	Load exponent (AC0 17-23 to FPAC 1-7)
FAB *	Compute absolute value (set sign of FPAC to zero)
FFAS *	Fix to AC (FPAC to AC)
FINT *	Integerize (FPAC)
FLAS *	Float from AC (AC to FPAC)
FNEG *	Negate
FNOM *	Normalize (FPAC)
FRDS	Floating-point round double to single
FRH *	Read high word (FPAC 0-15 to AC0 16-31)
FSCAL *	Scale floating point
WFFAD	Wide fix from FPAC
WFLAD	Wide float from AC

Table 3.1 Floating-point binary conversion instructions

*ECLIPSE C/350 compatible instruction

Table 3.2 lists the instructions that convert and move a fixed-point decimal between memory and a floating-point accumulator. Refer to the Fixed-Point Computing chapter for further information on the load and store integer instructions.

Instruction	Operation
WLDI	Convert a decimal and load into FPAC
WLDIX	Convert a decimal, extend, and load it into four FPAC's
WSTI	Convert FPAC data and load into memory
WSTIX	Convert the four FPAC's and load into memory

Table 3.2 Floating-point decimal conversion instructions

Move Instructions

All single-precision operations that specify an accumulator fetch the most significant 32 bits of the floating-point accumulator and ignore the least significant 32 bits. Upon completion of the specified operation, the processor returns the result to the most significant portion of the floating-point accumulator. The processor loads the least significant 32 bits of the floating-point accumulator with zeros. Table 3.3 lists the load and store floating-point accumulator instructions.

Instruction	Operation
FLDD *	Load floating-point double
FLDS *	Load floating-point single
FMOV *	Move floating point (FPAC to FPAC)
FSTD *	Store floating-point double
FSTS *	Store floating-point single
LFLDD	Load floating-point double
LFLDS	Load floating-point single
LFSTD	Store floating-point double
LFSTS	Store floating-point single
WFPOP	Wide floating-point pop
WFPSH	Wide floating-point push
XFLDD	Load floating-point double
XFLDS	Load floating-point single
XFSTD	Store floating-point double
XFSTS	Store floating-point single

Table 3.3 Floating-point data movement instructions

*ECLIPSE C/350 compatible instruction

Floating-Point Arithmetic Operations

To perform a floating-point arithmetic operation, the processor executes a floating-point arithmetic instruction. In executing the instruction, the processor

1. Appends one or two guard digits.
2. Aligns the mantissas (for addition and subtraction).
3. Calculates the result and normalizes it.
4. Adjusts the result by truncating or rounding it.
5. Stores the result in a floating-point accumulator or memory.

To increase the accuracy of the result, the processor appends one or two guard digits to the operands of both mantissas, before performing the arithmetic calculations. A *guard digit* is one hex digit (four bits) that initially contains a zero. The processor modifies the guard digits during the arithmetic calculations, which increases the accuracy of the result.

Appending Guard Digits

The processor appends the one or two guard digits to the least significant hex digit of both mantissas, depending on the RND flag (bit 8) in the floating-point status register. Use the load floating-point status register instruction (LFLST) to change the RND flag.

> **NOTE:** *The floating-point conversion and single precision store instructions (FINT, FSCAL, LFSTS, WFFAD, WFLAD, and XFSTS) ignore the RND flag. Refer to the individual instruction description in the Instruction Dictionary chapter for further information.*

When the RND flag equals zero, the processor appends one guard digit in preparation for truncating the mantissa of the intermediate result. When the RND flag equals one, the processor appends two guard digits in preparation for rounding the mantissa of the intermediate result. An *intermediate result* includes the exponent and the mantissa.

Aligning the Mantissas

For floating-point addition and subtraction, the processor first aligns the smaller mantissa to the larger mantissa. To align the mantissas, the processor takes the absolute value of the difference between the two exponents. If the difference equals nonzero, the processor shifts the mantissa with the smaller exponent to the right until the difference equals zero or until the processor shifts out the significant digits of the mantissa. The mantissas are aligned when the difference equals zero.

If the processor shifts out the significant digits, the operation is equivalent to adding zero to the number with the larger exponent. To shift out the significant digits, the processor must shift at least 7 or 8 hex digits for single precision (for truncating or rounding, respectively) or shift at least 15 or 16 hex digits for double precision.

Calculating and Normalizing the Result

The processor performs the floating-point arithmetic operation, determines by the rules of algebra the signs of the intermediate result, and then normalizes it. The processor normalizes an intermediate mantissa by shifting it left one hex digit at a time until the most significant hex digit represents a nonzero quantity. For each hex digit shifted left, the processor decrements the intermediate exponent by one. The processor zero fills the guard digit of the intermediate mantissa.

Truncating or Rounding the Result

As determined by the RND flag, the processor truncates or rounds the intermediate mantissa. When the RND flag equals zero, the processor *truncates* the intermediate mantissa by removing the guard digit. When the RND flag equals one, the processor *rounds* the intermediate mantissa by removing and analyzing the two guard digits.

When the two guard digits are

- Within the range of 0 to $7F_{16}$ inclusive, the intermediate result becomes the final result (without change).
- Equal to 80_{16} , the processor adds the least significant bit of the intermediate mantissa to the intermediate mantissa.

  The processor forces an even mantissa to be rounded down to the nearest integer and an odd mantissa to be rounded up to the nearest integer. If the processor rounded down or rounded up without an intermediate mantissa overflow, the operation produces the final result.
- Within the range of 81_{16} to FF_{16} inclusive, the processor adds 1_{16} to the intermediate mantissa.

  If the processor rounded up the intermediate mantissa without an overflow, the operation produces the final result.

If rounding up causes a mantissa overflow, the processor performs the following actions:

1. Shifts the intermediate mantissa right one hex digit.
2. Places 1_{16} into the most significant hex digit.
3. Adds one to the intermediate exponent.
4. Truncates the rightmost hex digit so that the intermediate mantissa is 24 or 56 bits, which becomes the final result.

Storing the Result

The processor stores the final result into the specified memory location or floating-point accumulator. The processor then checks for a possible exponent underflow or overflow. If no underflow or overflow exists, the instruction execution ends. If an underflow or overflow exists, the processor sets the appropriate error flag in the floating-point status register. The value of the exponent is undefined.

Arithmetic Instructions

Floating-point arithmetic instructions perform single and double precision addition, subtraction, multiplication, and division.

Addition

The processor adds the two mantissas together, producing an intermediate result. The processor determines the sign of the intermediate result from the signs of the two operands by the rules of algebra.

If the mantissa addition produces a carry out of the most significant bit, the processor shifts the intermediate mantissa to the right one hex digit and increments the exponent by one. If incrementing the exponent produces no exponent overflow and the intermediate mantissa equals a nonzero, the processor normalizes the intermediate mantissa, rounds or truncates it, and stores the final result in memory or in a floating-point accumulator.

If incrementing the exponent produces an exponent overflow, the processor sets the OVF error flag to one and terminates the instruction. If there is no mantissa overflow, but the intermediate mantissa contains all zeros, the processor places a true zero in memory or in a floating-point accumulator. Table 3.4 lists the floating-point add instructions.

Instruction	Operation
FAD *	Add double (FPAC to FPAC)
FAS *	Add single (FPAC to FPAC)
FAM *	Add double (memory to FPAC)
FAMS *	Add single (memory to FPAC)
LFAMD	Add double (memory to FPAC)
LFAMS	Add single (memory to FPAC)
XFAMD	Add double (memory to FPAC)
XFAMS	Add single (memory to FPAC)

Table 3.4 Floating-point addition instructions

*ECLIPSE C/350 compatible instruction

Subtraction

For floating-point subtraction, the processor temporarily complements the sign of the source mantissa and performs a floating-point addition. Upon completion, the difference is stored in the destination floating-point accumulator. Also the source mantissa returns to its original value when the source accumulator is different from the destination accumulator (facs ≠ facd). Table 3.5 lists the floating-point subtract instructions.

Instruction	Operation
FSD *	Subtract double (FPAC from FPAC)
FSS *	Subtract single (FPAC from FPAC)
FSMD *	Subtract double (memory from FPAC)
FSMS *	Subtract single (memory from FPAC)
LFSMD	Subtract double (memory from FPAC)
LFSMS	Subtract single (memory from FPAC)
XFSMD	Subtract double (memory from FPAC)
XFSMS	Subtract single (memory from FPAC)

Table 3.5 Floating-point subtraction instructions

*ECLIPSE C/350 compatible instruction

Multiplication

For floating-point multiplication, the processor multiplies one floating-point mantissa by the other floating-point mantissa, which produces an intermediate floating-point mantissa. The processor adds the two exponents, subtracts 64_{10} to maintain excess 64 notation, and produces an intermediate floating-point exponent. The processor then normalizes the intermediate mantissa, rounds or truncates it, and stores the final result. Table 3.6 lists the floating-point divide instructions.

Instruction	Operation
FMD *	Multiply double (FPAC by FPAC)
FMS *	Multiply single (FPAC by FPAC)
FMMD *	Multiply double (FPAC by memory)
FMMS *	Multiply single (FPAC by memory)
LFMMD	Multiply double (FPAC by memory)
LFMMS	Multiply single (FPAC by memory)
XFMMD	Multiply double (FPAC by memory)
XFMMS	Multiply single (FPAC by memory)

Table 3.6 Floating-point multiplication instructions

*ECLIPSE C/350 compatible instruction

Division

For floating-point division, the processor tests the divisor for zero. (The source location contains the divisor and the destination location contains the dividend.) If the divisor is zero, the processor sets the INV error flag to one, places error code zero in the INP bits, and the address of the instruction in the FPPC, and ends the instruction. If the divisor is nonzero, the processor compares the two mantissas. If the dividend mantissa is greater than or equal to the divisor mantissa, the processor aligns the two mantissas with the following actions:

1. Shifts the dividend mantissa to the right one hex digit.
2. Places 0_{16} into the most significant digit of the dividend mantissa.
3. Adds one to the dividend exponent.

When the dividend mantissa is less than the divisor mantissa, the processor performs the following actions:

1. Divides the mantissas, which produces an intermediate floating-point mantissa.
2. Subtracts the divisor exponent from the dividend exponent, and adds 64_{10} to the difference (maintaining the excess 64 notation), which produces an intermediate floating-point exponent.
3. Normalizes and rounds or truncates the intermediate mantissa, which produces the final result (exponent and mantissa).
4. Stores the final result in memory or a floating-point accumulator.

Table 3.7 lists the floating-point divide instructions.

Instruction	Operation
FDD *	Divide double (FPAC by FPAC)
FDS *	Divide single (FPAC by FPAC)
FDMD *	Divide double (FPAC by memory)
FDMS *	Divide single (FPAC by memory)
FHLV *	Halve (FPAC/2)
LFDMD	Divide double (FPAC by memory)
LFDMS	Divide single (FPAC by memory)
XFDMD	Divide double (FPAC by memory)
XFDMS	Divide single (FPAC by memory)

Table 3.7 Floating-point division instructions

*ECLIPSE C/350 compatible instruction

Skip Instructions

A skip instruction tests the result of an operation for a specific condition and (except for FCMP) directs the processor to skip the word or to execute the word after the skip instruction. The FCMP instruction compares two floating-point numbers and sets the Z and N status flags reflecting the relationship. You can then use the FSGT, FSEQ, and FSLT skip instructions to test the status flags.

Table 3.8 lists the floating-point skip on condition instructions. When a skip occurs, the processor increments the program counter by one and executes the second word after the skip instruction.

NOTE: *Be sure that a skip does not transfer control to the middle of a 32-bit or larger instruction.*

Instruction	Operation
FCMP *	Compare two floating-point numbers (set N and Z)
FSEQ *	Skip on zero (Z = 1)
FSGE *	Skip on greater than or equal to zero (N = 0)
FSGT *	Skip on greater than zero (N and Z = 0)
FSLE *	Skip on less than or equal to zero (N and Z = 1)
FSLT *	Skip on less than zero (N = 1)
FSND *	Skip on no zero divide (DVZ = 0)
FSNE *	Skip on nonzero (Z = 0)
FSNER *	Skip on no error (ANY = 0)
FSNM *	Skip on no mantissa overflow (MOF = 0)
FSNO *	Skip on no overflow (OVF = 0)
FSNOD *	Skip on no overflow and no zero divide (OVF and DVZ = 0)
FSNU *	Skip on no underflow (UNF = 0)
FSNUD *	Skip on no underflow and no zero divide (UNF and DVZ = 0)
FSNUO *	Skip on no underflow and no overflow (UNF and OVF = 0)

Table 3.8 Floating-point skip on condition instructions

*ECLIPSE C/350 compatible instruction

Intrinsic Instruction Set

The optional Intrinsic Instruction Set (IIS) performs trigonometric and logarithmic functions, exponentiation, and square root evaluation on single-precision and double-precision data.

These instructions assume the argument to be operated on will be in FPAC0 and the answer will be returned in FPAC0. For two-argument instructions (such as WFATN2 and WFPWR), FPAC1 will be used for the second argument. Upon instruction completion, the contents of the remaining floating-point accumulators are undefined.

All floating-point inputs are assumed to be normalized. Any input with a zero mantissa is assumed also to have a zero sign bit and an all zero exponent (True Zero).

The trigonometric instructions (sine, cosine, tangent) require a floating-point input in radians while the inverse trigonometric instructions (arcsine, arccosine, arctangent) return the result in radians.

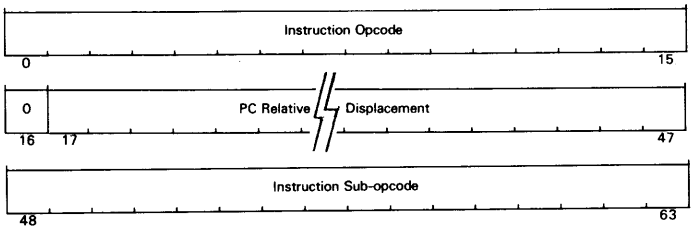
The Z and N flags of the FPSR are always updated by IIS instructions to reflect the result -- either zero or negative.

An IIS instruction can cause floating-point traps, if traps are enabled, for either invalid input or for a result that overflows or underflows.

If an invalid normalized number or an illegal argument is used as input to an IIS instruction, a trap will occur. An illegal argument causes the processor to place the instruction address in the FPSR floating-point program counter (FPPC), set the INV bit in the FPSR to one and place an error code in FPSR bits 28-31 (INP). The error code indicates what type of input error occurred. For example, a negative value input to a square root (WFSQR) instruction will cause an invalid input argument trap with the error code two returned to the INP bits. The FPSR description in the "Faults and Status" section of this chapter explains the various codes and their meanings.

If the result of an IIS instruction has overflowed or underflowed, a floating-point trap will occur with the relevant error bits (OVF or UNF) updated in the FPSR. Overflow and underflow errors behave identically to the standard floating-point instruction errors.

The IIS instructions have a displacement coded with each instruction. The format for these four-word instructions is:



Machines which implement these instructions in hardware ignore the displacement (however, invalid displacement addresses can cause unspecified actions -- refer to the functional characteristics manual for your machine). The coded displacement is a PC-relative, non-indirectable address of a routine in an optional run time library in the current ring that will emulate the function of the instruction if hardware support does not exist.

All IIS instructions are interruptible and resumable. Table 3.9 lists the intrinsic instructions.

Instruction	Function
WFACOSD	Arccosine double
WFACOSS	Arccosine single
WFASIND	Arcsine double
WFASINS	Arcsine single
WFATAND	Arctangent double
WFATANS	Arctangent single
WFATN2D	Arctangent double (two-accumulator)
WFATN2S	Arctangent single (two-accumulator)
WFCOSD	Cosine double
WFCOSS	Cosine single
WFEXPD	Exponential double
WFEXPS	Exponential single
WFLG2D	Binary logarithm double
WFLG2S	Binary logarithm single
WFLNGD	Natural logarithm double
WFLNGS	Natural logarithm single
WFLOGD	Common logarithm double
WFLOGS	Common logarithm single
WFPWRD	Power double
WFPWRS	Power single
WFSIND	Sine double
WFSINS	Sine single
WFSQRD	Square root double
WFSQRS	Square root single
WFTAND	Tangent double
WFTANS	Tangent single

Table 3.9 Floating-point intrinsic instructions

Faults and Status

The processor checks for a floating-point fault and for the mantissa status after executing a floating-point instruction. The processor stores the result in a 64-bit floating-point status register. When the processor detects a floating-point fault (overflow or underflow), the processor sets the appropriate floating-point status register bits.

For the processor to service the fault, it must first determine the state of the trap enable (TE) mask (bit 5 of the floating-point status register). If the TE mask equals zero, the processor continues normal program execution with the next sequential instruction. Program flow remains unchanged. If the TE mask equals one, the processor disrupts normal program execution by performing an indirect jump to the floating-point fault handler to service the fault. Refer to the Program Flow Management chapter for further information on fault handling.

You access the floating-point status register with instructions that initialize the register or that test the register bits. The Skip section lists the instructions that test the bits. Table 3.10 lists the instructions that initialize the register and that store or load the register contents.

Instruction	Operation
FCLE *	Clear errors (FPSR)
FTD *	Floating-point trap disable (resets TE)
FTE *	Floating-point trap enable (sets TE)
LFLST	Load FPSR
LFSST	Store FPSR
WFPSH	Push floating-point state
WFPOP	Pop floating-point state

Table 3.10 Floating-point status instructions

*ECLIPSE C/350 compatible instruction

The floating-point status register contains fault flags (ANY, OVF, UNF, INV, MOF, and TE), mantissa status (Z and N), rounding (RND), a floating-point identification (ID), invalid input argument indicator (INP) and a floating-point program counter (FPPC). Figure 3.2 shows the format of the floating-point status register.

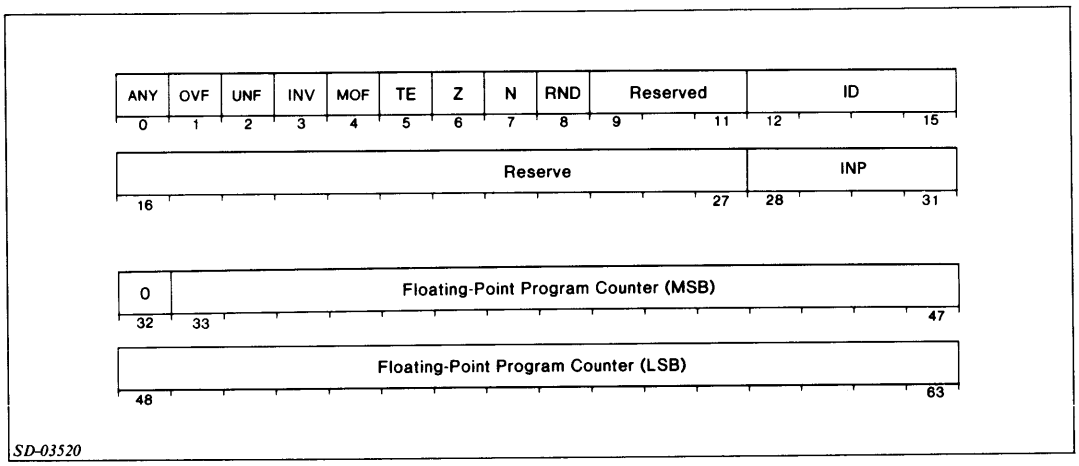


Figure 3.2 Floating-point status register format

where

ANY The ANY bit is an error status flag.

The processor sets the ANY flag to one when it sets either the OVF, UNF, INV, or MOF flag to one.

OVF The OVF bit is the exponent overflow flag.

The processor sets the OVF flag while executing a floating-point instruction, and an exponent overflow occurs. The result is correct except the exponent is 128 too small.

UNF The UNF bit is the exponent underflow flag.

The processor sets the UNF flag while executing a floating-point instruction, and an exponent underflow occurs. The result is correct except the exponent is 128 too large.

INV — The INV bit is the input argument error flag.

The processor sets the INV flag while attempting to execute an instruction with an invalid argument as input. If the INV bit is one, the INP bits will contain a further definition of the input error. The processor then aborts the operation and the operands remain unchanged.

NOTE: *The previous definition of this flag was Divide by Zero (DVZ) which has been expanded to include other input argument errors.*

MOF — The MOF bit is the mantissa overflow flag.

The processor sets the MOF flag while executing a floating-point instruction when it detects a mantissa overflow. If it occurs during a FSCAL instruction, the processor shifts out the most significant bit. If it occurs during a FFAS, FFMD, or WFFAD instruction, the result is too large and the processor truncates the result before storing it.

TE — The TE bit is the trap enable mask.

The processor or you enable floating-point fault detection and servicing by setting the TE mask to one. You can set the TE mask to one with the FTE instruction.

Unless your system is running with a parallel floating-point unit, the processor does not save or restore the status of the TE mask when going to or returning from a subroutine or fault handler. Refer to the Processor Status Description in the Fixed-Point Computing chapter for further information.

For the processor to detect and to service a fault, the TE mask must be set to one before the processor sets the ANY flag to one. If TE is set to one, a one in any of bits 1 through 4 will result in a floating-point trap, except where noted.

Z — The Z bit is the true zero flag.

The processor sets the Z flag if the result of executing a floating-point instruction produces a true zero.

N — The N bit is the negative flag.

The processor sets the N flag if the result of executing a floating-point instruction produces a value less than zero.

RND — The RND bit is the round flag.

You set the RND flag (with the LFLST and WFPOP instructions), which directs the processor to round (RND = 1) or to truncate (RND = 0) the intermediate result of executing a floating-point instruction.

Reserve — The reserve bits 9-11 are processor specific.

ID — The ID code is a floating-point identification code that reflects the floating-point revision.

Reserve — The reserve bits 16-27 are processor specific.

INP — The INP bits contain an indicator for an invalid input argument. The definition of the INP bits is dependent on the setting of the invalid input argument (INV) bit.

If the INV bit is zero, the INP bits are undefined.

If the INV bit is one, the value contained in the INP bits indicates an attempt to use an invalid input. A code value greater than zero applies to the floating-point Intrinsic Instruction Set (IIS) option. The presently defined values are:

Code in Bits 29-31	Instruction in Error	Description
0	**FDS, FDD, FDMS, FDMD, XFDMS, XFDMD, LFDMS, LFDMD**	Attempt to execute a floating-point divide instruction with a divisor equal to 0.
1	**WFLOGS, WFLOGD WFLG2S, WFLG2D WFLNGS, WFLNGD**	FPAC0 contains a value less than or equal to 0.
2	**WFSQRS, WFSQRD**	FPAC0 contains a value less than 0.
3	**WFASINS, WFASIND WFACOSS, WFACOSD**	The absolute value of FPAC0 is greater than 1.
4	**WFPWRS, WFPWRD**	The value in FPAC0 is less than 0 and the value in FPAC1 is not equal to 0, or the value in FPAC0 is equal to 0, and the value in FPAC1 is less than or equal to 0.
5	**WFEXPS, WFEXPD**	The number in FPAC0 will cause an overflow.
6	**WFTANS, WFTAND**	Special overflow for WFTAN (numbers in FPAC0, which are integer multiples of values near PI/2, will cause an overflow).
7	**WFATN2S, WFATN2D**	The number in FPAC1 equals 0.

NOTE: *If floating-point traps are disabled (TE bit equals 0), more than one invalid input argument error may occur before floating-point traps are again enabled. In this case, the INP bits will contain the error code for the FIRST instruction which caused an invalid input argument error.*

When the processor detects an invalid input error, the contents of the floating-point accumulators are unmodified and Carry and overflow are unchanged.

0 — Bit 32 is processor specific.

Floating-Point Program Counter — The floating-point program counter (FPPC) contains the address of the first floating-point instruction to set an error bit in the FPSR (after an FCLE or IORST) unless specifically set by a Load Floating-Point State instruction. FPPC is undefined if ANY=0.

NOTES: *All reserved bits in the FPSR must be zero. The OVF, UNF, INV, and MOF status bits are cumulative. These bits are only cleared by FPSR loads which restore them, or the explicit clear instructions (such as FCLE).*

Chapter 4
Stack Management

Overview

A *stack* is a series of consecutive locations in memory. In the simplest form, stack instructions add items in sequential order to the top of the stack and retrieve them in the reverse order. A program can access several stack areas, but can use only one stack area at any time. The processor, using the push-down stack concept, pushes (stores) data onto the stack and pops (retrieves) data from it in the reverse order.

For instance, the processor can push or pop the contents of up to four accumulators with the WPSH or WPOP instruction. In addition, the processor can push a return block for a subroutine call, an I/O interrupt request, or a fault. Then a return block would be popped upon returning from the call, interrupt, or fault handler.

The 32-bit processor provides facilities for wide and narrow stack operations. The wide stack is a series of double words, and supports 32-bit programs. The system includes four 32-bit stack management registers to manage the wide stack operations. The narrow stack is a series of single words, and supports 16-bit programs (for ECLIPSE C/350 program development and upward program compatibility). The system uses three words in reserved memory to manage the narrow stack operations.

The remainder of the chapter presents the wide stack operations and instructions. Refer to the ECLIPSE C/350 Compatibility chapter for further information on the narrow stack. The Program Flow Management chapter presents wide and narrow stack fault handling.

Wide Stack Operations

Each segment contains a set of wide stack parameters. The processor manages the stack parameters in the current segment with four 32-bit stack registers. You can modify the contents of the stack registers with instructions that move data between an accumulator and a stack register.

When transferring program control to another segment, the processor stores the contents of the stack registers in page zero of the current segment and initializes the contents of the stack registers from page zero of the destination segment. You can modify the stack parameters with memory reference instructions to the appropriate page zero locations.

NOTE: *A program must not refer to the stack parameters in page zero of the current segment.*

Figure 4.1 shows the four stack parameters. Items (1) and (2) identify the lower and upper stack limits, which define the locations that the stack occupies. Items (3) and (4) identify the wide stack pointer and the wide frame pointer, which address the data in the stack.

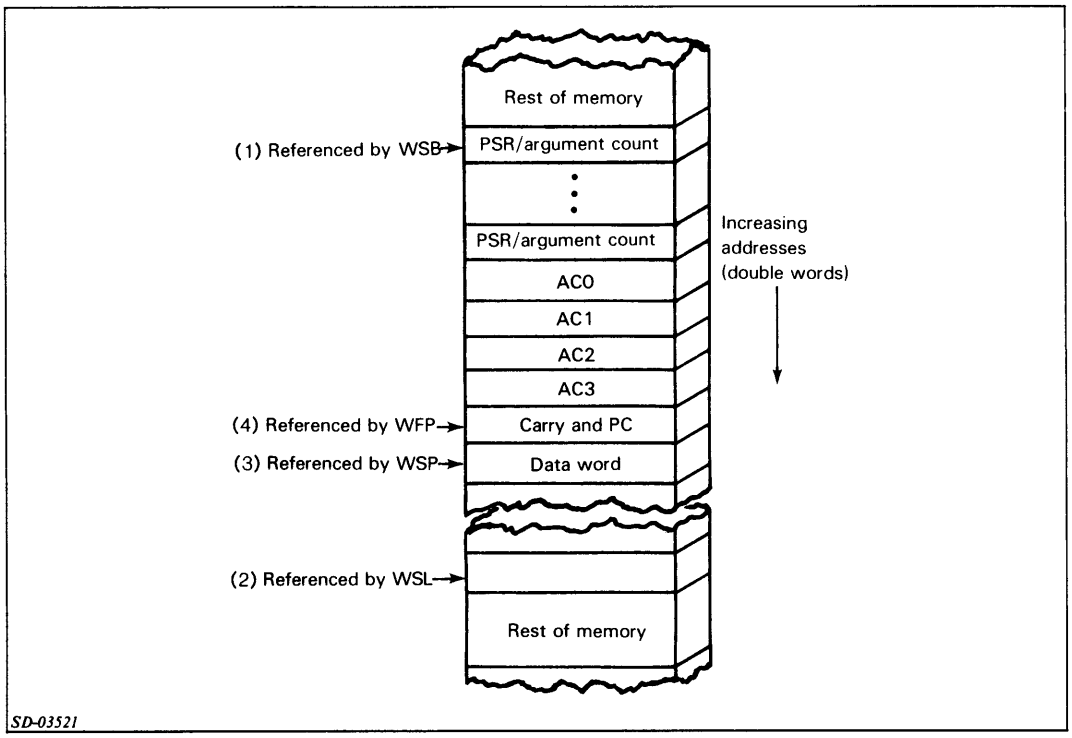


Figure 4.1 A typical wide stack

Wide Stack Registers

You should initialize the contents of the wide stack registers to address locations that are aligned on double word boundaries (even addresses). Stack operations for some stack instructions can be slower if the registers contain odd addresses.

Wide Stack Base

The wide stack base (WSB) defines the lower limit of the wide stack. When you initialize a wide stack, the wide stack base must be one double word below the actual address of the first double word in the wide stack.

The processor uses the wide stack base contents when it pops data from the wide stack. For instance when returning from a subroutine, the processor pops a wide return block and then checks for a wide stack underflow. If the wide stack pointer value is less than the wide stack base value, an underflow condition exists. Refer to the Wide Stack Faults section for further information on handling an underflow fault.

Wide Stack Limit

The wide stack limit (WSL) defines the upper limit of the wide stack. When you initialize a wide stack, the wide stack limit must be 24 double words below the actual address of the last double word in the wide stack.

The processor pushes one or more double words onto the wide stack (such as a wide return block when calling a subroutine), and then for most operations checks for a stack overflow fault. (However, the processor checks for overflow before pushing data onto the stack when using the wide save instructions (WSAVR, WSAVS, WSSVR, or WSSVS) and when crossing to a subroutine in a lower-numbered segment.)

To check for a wide stack overflow fault, the processor compares the wide stack pointer contents to the wide stack limit contents. If the wide stack pointer contents are greater than the wide stack limit contents, an overflow condition exists. Refer to the Wide Stack Faults section for further information on handling an overflow fault.

Wide Stack Pointer

The wide stack pointer (WSP) addresses the top location of the wide stack; either the location of the last word placed onto the stack or the next word available from the stack. When you initialize a wide stack, set the wide stack pointer equal to the address in the wide stack base register.

To push a double word, the processor increments the wide stack pointer by two and stores a double word onto the stack. A pop operation retrieves one or more double words from the wide stack and decrements the wide stack pointer by two for each double word it pops.

NOTE: *The area between the wide stack pointer and the wide stack limit can be modified by the processor. For example, the WEDIT instruction may store temporary WEDIT data.*

Wide Frame Pointer

The wide frame pointer (WFP) -- unchanged by push and pop operations -- defines a reference point in the wide stack. When you set up a wide stack, initialize the wide frame pointer with the same value as the wide stack pointer to preserve the original value of the wide stack pointer.

The processor stores and resets the value of the wide frame pointer when entering or leaving subroutines. Thus, the wide frame pointer identifies the boundary between words placed on the wide stack before a subroutine call, and between words placed on the wide stack during a subroutine execution. Using the wide frame pointer as a reference, the processor can move back into the wide stack and retrieve arguments stored there by a preceding routine.

Wide Stack Register Instructions

The instructions listed in Table 4.1 load (or initialize) a wide stack register with data from an accumulator, or store data into an accumulator from a wide stack register. In addition, when the LCALL, WRTN, or XCALL instruction transfers program control to another segment, the processor initializes all four wide stack registers.

Instruction	Operation
LDAFP	Load accumulator with the WFP register contents
LDASB	Load accumulator with the WSB register contents
LDASL	Load accumulator with the WSL register contents
LDASP	Load accumulator with the WSP register contents
STAFP	Store accumulator in the WFP register
STASB	Store accumulator in the WSB register
STASL	Store accumulator in the WSL register
STASP	Store accumulator in the WSP retister
WMSP	Wide modify WSP register

Table 4.1 Wide stack register instructions

Wide Stack Data Instructions

The wide stack data instructions access a double word or a block of double words. All the wide stack data instructions increment or decrement the wide stack pointer. Instructions that access a double word, modify the wide stack pointer by two. Instructions that access a block of double words modify the wide stack pointer by four or more (depending upon the size of the data block or return block). The instructions in Table 4.2 access a double word or a block of double words.

Instruction	Operation
DSZTS	Decrement the double word addressed by WSP (skip if zero)
ISZTS	Increment the double word addressed by WSP (skip if zero)
LDATS	Load accumulator with double word addressed by WSP
LPEF	Push address
LPEFB	Push byte address
LPSHJ	Push jump to subroutine (pop with WPOPJ)
STATS	Store accumulator into double word addressed by WSP
WFPOP	Wide floating-point pop
WFPSH	Wide floating-point push
WPOP	Wide pop accumulators (push with WPSH)
WPOPJ	Wide pop PC and jump (push with LPSHJ or XPSHJ)
WPSH	Wide push accumulators (pop with WPOP)
XPEF	Push address
XPEFB	Push byte address
XPSHJ	Push jump to subroutine (pop with WPOPJ)

Table 4.2 Wide stack double-word access instructions

The instructions in Table 4.3 push or pop a return block. Although the return block can take several forms, it usually consists of six double words (see Table 4.4).

Instruction	Operation
BKPT	Breakpoint handler (return from breakpoint handler with PBX)
LCALL	Call subroutine (return from call with WRTN)
PBX	Pop block and execute (return from breakpoint handler)
WPOPB	Wide pop block
WRSTR	Wide restore from an interrupt
WRTN	Wide return via wide save (WSAVR, WSAVS, WSSVR, and WSSVS)
WSAVR	Wide save/reset overflow mask (used with LCALL and XCALL)
WSAVS	Wide save/set overflow mask (used with LCALL and XCALL)
WSSVR	Wide special save/reset overflow mask (used with LJSR & XJSR)
WSSVS	Wide special save/set overflow mask (used with LJSR & XJSR)
WXOP	Extended operation (return with WPOPB; used to expand instruction set)
XCALL	Call subroutine (return from call with WRTN)

Table 4.3 Wide stack return block instructions

Word Number in Block Pushed	Word Number in Block Popped	Contents
1	12	PSR
2	11	All zeros or an argument count from LCALL or XCALL
3-4	9-10	AC0
5-6	7-8	AC1
7-8	5-6	AC2
9-10	3-4	AC3 = Old WFP
11-12	1-2	Bit 0 = CARRY; Bits 1-31 = PC return address

Table 4.4 Standard wide return block

The Instruction Dictionary chapter presents the subroutine return block with a subroutine instruction description. The Program Flow Management chapter identifies the return blocks for the nonprivileged faults, while the Device Management chapter presents the return block for an I/O interrupt. Then, the Memory and System Management chapter identifies the return blocks for privileged operations.

Initializing A Wide Stack

Figure 4.2 illustrates assembler code for initializing a wide stack. The stack resides in locations 256_{10} through 355_{10}. The processor detects a stack overflow 17 double words before the actual end of the stack.

```
        .NREL
BASE:   .BLK    66.         ; Reserve 66 words for the wide stack
ENDZ:   .BLK    34.         ; Reserve 34 words for wide stack end zone
          .
          .
          .

        XLEF    0,BASE
        STASB   0           ; Initialize WSB
        XLEF    0,ENDZ      ; Initialize WSL for a stack
        STASL   0           ;   overflow when WSP = BASE+66
        XLEF    0,BASE-2
        STASP   0           ; Initialize WSP
        STAFP   0           ; Initialize WFP
          .
          .
          .
        XPEFB   BYTZ*2      ; Calculate and store the byte address
          .                     for BYTZ
          .
          .
```

SD-03522

Figure 4.2 Sample code for initializing a wide stack

Figure 4.3 illustrates the result of executing the assembler code in Figure 4.2. The XPEFB instruction calculates and pushes a byte address onto the stack.

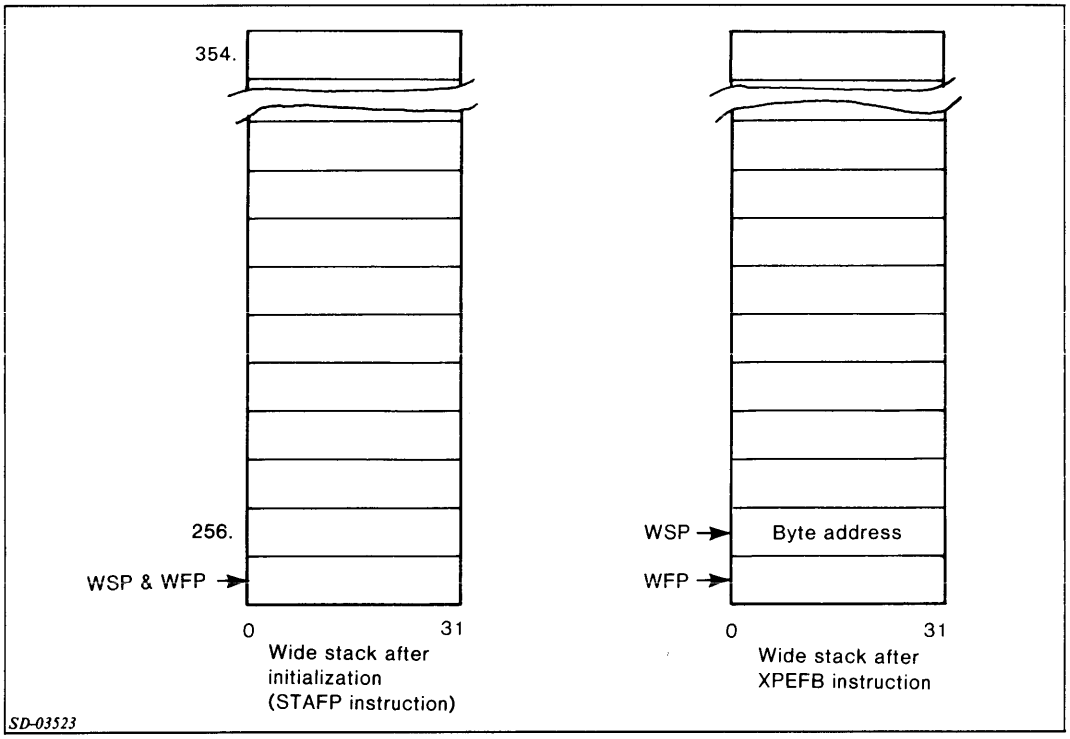


Figure 4.3 Example of wide stack operations

Wide Stack Faults

Stack overflow and underflow are stack faults. Stack overflow occurs when a program pushes data into the area beyond that allocated for the stack. Stack underflow occurs when a program pops data from the area beyond that allocated for the stack. Once detected, the processor always processes a stack fault.

After pushing data onto the stack, the processor checks for a stack overflow by comparing the value of the wide stack pointer to the value of the wide stack limit. If the value of the wide stack pointer is greater, then a stack overflow exists. Loading the value 37777777777_8 into the wide stack limit register disables wide stack overflow fault detection.

After popping data from the stack, the processor checks for a stack underflow by comparing the value of the wide stack pointer to the value of the wide stack base. If the value of the wide stack pointer is less, then a stack underflow exists. Loading the value 20000000000_8 into the wide stack base register disables wide stack underflow fault detection.

Table 4.5 lists the instructions that push or pop one or more double words onto the wide stack. Table 4.5 also lists for the instructions the number of words required beyond the wide stack limit for a stack fault return block. Refer to the Program Flow Management chapter for a description of stack fault servicing.

Instruction	Description	Double Words	
		Pushed or (Popped)	Required Beyond WSL for Stack Fault
ADD, etc.	Arithmetic with OVK enabled	0	11
FAD, etc.	Arithmetic with TE enabled	0	11
BKPT	Breakpoint handler	6	11
LCALL	Subroutine call	1	6
LPEF	Push address	1	6
LPEFB	Push byte address	1	6
LPSHJ	Push jump	1	6
PBX	Pop block and execute	(6)	5
WEDIT	Wide edit	16	27
WFPOP	Wide floating-point pop	(10)	5
WFPSH	Wide floating-point push	10	15
WPOP	Wide pop accumulators	(1-4)	5
WPOPB	Wide pop block	(6)	5
WPOPJ	Wide pop PC and jump	(1)	5
WPSH	Wide push accumulators	1-4	9
WRSTR	Wide restore	(10)	5
WRTN	Wide return	(6)	5
WSAVR	Wide save/reset OVK	5	10
WSAVS	Wide save/set OVK	5	10
WSSVR	Wide special save/reset OVK	6	11
WSSVS	Wide special save/set OVK	6	11
WXOP	Extended operation	6	11
XCALL	Subroutine call	1	6
XPEF	Push address	1	5
XPEFB	Push byte address	1	5
XPSHJ	Push jump to subroutine	1	5
XVCT	Vector on I/O interrupt	6	11

Table 4.5 Multiword wide stack instructions

Chapter 5
Program Flow Management

Overview

The Program Flow Management chapter explains program flow, related instruction groups, transferring program control to another segment, and handling faults.

Program Flow

The program counter specifies the logical address of the instruction to execute. Thus, it controls the sequence of executing the instructions. Address wraparound occurs within the current segment since only bits 4 through 31 take part in incrementing the program counter.

To address the next instruction (for normal program flow), the processor increments the program counter

- By one, when executing a one word instruction (such as NADI).
- By two, when executing a two word instruction (such as NADDI).
- By three, when executing a three word instruction (such as LNADI).
- By four, when executing a four word instruction (such as LCALL).

Any of the following events alter the normal program flow sequence.

- Executing the XCT instruction.
- Executing a jump instruction.
- Executing a skip instruction.
- Executing a subroutine call or return instruction.
- Detecting a fault.
- Detecting an I/O interrupt request.

The next section explains the XCT, jump, skip, and subroutine call or return instruction. Refer to the Device Management chapter for I/O interrupt processing.

Related Instruction Groups

Execute Accumulator

The execute accumulator instruction (XCT) executes bits 16-31 of an accumulator as an instruction. After executing the accumulator contents, program flow continues with one of the following locations.

- The first location after the XCT instruction.
- The second location after the XCT instruction, if the accumulator is the first of a two word instruction.
- The effective address, if the accumulator contains a jump or skip instruction.

Jump

A jump instruction loads the effective address into the program counter. Program flow continues at the effective address. A jump instruction does not save a return address. The jump instructions are listed in Table 5.1.

Instruction	Operation
LDSP	Dispatch
LJMP	Jump (with long displacement)
WBR	Branch (PC relative jump)
XJMP	Jump (with extended displacement)

Table 5.1 Jump instructions

Skip

A skip instruction jumps the first word after the skip instruction, and executes the second word as an instruction. To perform the skip, the processor adds one to the program counter. For most skip instructions, the processor first tests a machine condition or status, and based on the test results, it executes the first or second word as an instruction.

When you use a skip instruction, be sure that the skip does not transfer control to the middle of a two (or more) word instruction. For instance, the first two lines of code in Figure 5.1 perform an illegal skip because the program counter contains the address of the first word of the LFDMD two-word displacement. The last three lines of code in Figure 5.1 perform the skip properly.

```
        FSEQ                :skip on zero
        LFDMD   0.@OPAND    :floating-point divide with a two-word displacement
          .
          .
          .
        FSNE                :skip on nonzero asnd execute the LFDMD instruction
        WBR     NEXT        :zero -- skip the LFDMD instruction
        LFDMD    0.@OPAND   :floating-point divide with a two-word displacement
      NEXT:
```

SD-03524

Figure 5.1 Illegal and legal skip instruction sequences

Certain skip instructions modify the program counter by one or more words. For instance, the FNS instruction never skips the next instruction, and the FSA instruction always skips the next instruction. Table 5.2 lists the instructions.

Instruction	Operation
FNS *	No skip
FSA *	Skip always
LNDO	Narrow do until greater than
LWDO	Wide do until greater than
XNDO	Narrow do until greater than
XWDO	Wide do until greater than
NBStc	Narrow search queue backward
NFStc	Narrow search queue forward
WBStc	Wide search queue backward
WFStc	Wide search queue forward

Table 5.2 Skip instructions

*ECLIPSE C/350 compatible instruction

A DO-loop instruction (LNDO, LWDO, XNDO, and XWDO) increments a loop variable by one and then compares it to a value in a specified accumulator. The processor executes the

- First instruction of the DO-loop sequence when the incremented variable equals (or remains less than) the value.
- Instruction following the DO-loop sequence when the incremented variable becomes greater than the value.

The processor skips the DO-loop sequence of instructions

1. By adding one and the termination offset (for skipping the DO-loop sequence) to the program counter value (for a PC relative skip).
2. By loading the sum into the program counter.

For example, the lines of code in Figure 5.2 perform a valid DO-loop sequence.

```
                WXOR    0,0                     ; Zero AC0
                XNSTA   0,TEMP,3                ; Initialize variable (TEMP) to zero
                NLDAI   10,1                    ; Initialize constand (AC1) to ten
LOOP:           XNDO    1,TEMP,3,TERM - LOOP    ; Increment TEMP and compare to AC1
                  .                             ; Begin here to perform DO-loop
                  .                             ;       .
                  .                             ;       .
                                                ;       .
TERM:           WBR     LOOP                    ; Continue DO-loop until TEMP > AC1
                  .                             ; Terminate DO-loop when TEMP > AC1
                  .                             ;
                  .
```

SD-03525

Figure 5.2 DO-loop instruction sequence

A search queue instruction (NBStc, NFStc, WBStc, and WFStc) skips one, two, or three locations when an explicit queue element exists. Refer to the Queue Management chapter for more information on the search queue instructions.

In addition to the program flow and search queue instructions, additional skip instructions are available for fixed-point, floating-point, and I/O operations. For more information, refer to the following chapters.

- Fixed-Point Computing chapter for the fixed-point skip instructions.
- Floating-Point Computing chapter for the floating-point skip instructions.
- Device Management chapter for the I/O skip instructions.

Subroutine

A subroutine call sequence (except WEDIT) pushes a wide return block onto the wide stack and loads the effective address into the program counter. Program flow continues with the effective address in the program counter. (The WEDIT instruction transfers control to an edit subprogram without changing the program counter.)

> **NOTE:** *To pass arguments to the subroutine, push the arguments onto the stack before jumping (LJSR or XJSR) or calling (LCALL or XCALL) the subroutine.*

A subroutine return instruction (except WEDIT) pops the wide return block from the wide stack. Thus, restoring the carry, the program counter, and the accumulators. Program flow continues with the instruction following the subroutine call. (A WEDIT subprogram instruction returns program control to the instruction following the WEDIT instruction.) Table 5.3 lists the subroutine, save, and return instructions. Table 5.4 illustrates the relationships between the various subroutine instructions.

Instruction	Operation
BKPT	Breakpoint handler
LCALL	Call subroutine
LJSR	Jump to subroutine
LPSHJ	Push jump
PBX	Pop block and execute
WEDIT	Wide edit of alphanumeric
WPOPB	Wide pop block
WPOPJ	Wide pop PC and jump
WRTN	Wide return
WSAVR	Wide save/reset overflow mask
WSAVS	Wide save/set overflow mask
WSSVR	Wide special save/reset overflow mask
WSSVS	Wide special save/set overflow mask
WXOP	Wide extended operation
XCALL	Call subroutine
XJSR	Jump to subroutine
XPSHJ	Push jump

Table 5.3 Subroutine instructions

Call Instruction or Sequence	Segment Crossing Permitted	Associated Save Instruction	Return Instruction
BKPT	no		**PBX/WPOPB***
LCALL	yes	**WSAVR**	**WRTN**
	yes	**WSAVS**	**WRTN**
LJSR	no	**WSSVR**	**WRTN**
	no	**WSSVS**	**WRTN**
LPSHJ	no		**WPOPJ**
WEDIT	no		**DEND**
WXOP	no		**WPOPB**
XCALL	yes	**WSAVR**	**WRTN**
	yes	**WSAVS**	**WRTN**
XJSR	no	**WSSVR**	**WRTN**
	no	**WSSVS**	**WRTN**
XPSHJ	no		**WPOPJ**

Table 5.4 Sequence of subroutine instructions

*Use the BKPT/WPOPB instruction sequence when removing the BKPT instruction before returning from the breakpoint handler.

The *Breakpoint* (BKPT) instruction pushes a wide return block and transfers program control to the breakpoint handler. The Pop Block and Execute PBX instruction returns program control from the breakpoint handler.

Before executing the BKPT instruction, you must first store in memory the one-word opcode from the location that the BKPT instruction will occupy. Then, store the BKPT instruction in that one-word location.

When the processor executes the BKPT instruction, it pushes a wide return block onto the current stack and jumps to the breakpoint handler. When returning program control, the breakpoint handler must load the one-word opcode from memory into AC0. Then it executes the PBX instruction, which

1. Temporarily disables the interrupt system for one instruction execution;
2. Temporarily saves the one-word opcode in AC0 bits 16-31 and performs a WPOPB;
3. Temporarily replaces the BKPT instruction with the temporarily saved one-word opcode and then continues normal program flow.

If an interrupt occurs while the processor is executing the saved instruction (PC points to the BKPT instruction), the processor sets the IXCT flag in the PSR and pushes the opcode of the saved instruction on the wide stack. Upon returning from the interrupt handler, the BKPT instruction tests the IXCT flag. If the flag is set, the BKPT instruction resets the flag to 0, pops the saved opcode of the interrupted instruction off the wide stack, and executes it.

A *jump to a subroutine* (LJSR or XJSR) instruction transfers program control to a subroutine in the current segment. The LJSR or XJSR instruction stores the return address and transfers program control to the effective address. As the first instruction of the subroutine, a wide special save (WSSVR or WSSVS) instruction pushes a standard wide return block onto the wide stack. As the last instruction of the subroutine, the wide return (WRTN) instruction returns program control from the subroutine.

A *push and jump to a subroutine* (LPSHJ or XPSHJ) instruction pushes a return address onto the wide stack and transfers program control to the effective address in the current segment. As the last instruction of the subroutine, the WPOPJ instruction returns program control from the subroutine.

A *call to a subroutine* (LCALL or XCALL) instruction transfers program control to a subroutine in the current segment or in another segment and pushes (or copies) a double word onto the destination wide stack. As the first instruction of the subroutine, a wide save (WSAVR or WSAVS) instruction pushes a standard wide return block onto the wide stack in the destination segment. As the last instruction of the subroutine, the wide return (WRTN) instruction returns program control from the subroutine. (Refer to the next section for a complete description of transferring program control to another segment.)

Return Block

Although a wide return block can take several forms, it usually consists of six double words, as shown in Table 5.5. The fifth double word contains the contents of AC3 (for a BKPT or WXOP) or the previous wide frame pointer (for XCALL or LCALL and WSAVS or WSAVR). Bit 0 of the sixth double word contains the CARRY flag; bits 1-31 always contain the contents of the program counter.

Word Number in Block Pushed	Word Number in Block Popped	Contents
1	12	PSR
2	11	All zeros or an argument count from LCALL or XCALL
3-4	9-10	AC0
5-6	7-8	AC1
7-8	5-6	AC2
9-10	3-4	AC3 or old wide frame pointer
11-12	1-2	Bit 0 = CARRY flag Bits 1-31 = return address

Table 5.5 Standard wide return block

Example with wide stack operations

The following explanation illustrates the effects of a jump to subroutine (XJSR) instruction on a wide stack. The jump occurs within the current segment. The routine passes arguments to the subroutine by pushing the arguments onto the stack before executing the XJSR instruction.

Figure 5.3 illustrates the two lines of processor-related assembler code for beginning and ending a subroutine. The first instruction of the subroutine is a wide special save instruction (WSSVS) and the last instruction of the subroutine is a wide return instruction (WRTN).

The second instruction of the subroutine (XPEF) is provided to further illustrate the wide stack operations.

```
                .
                .
                .
SUB:    WSSVS   0       ; Save a wide return block
HERE:   XPEF    HERE    ; Calculate and push this address into the
                .       ; wide stack
                .
                .

                WRTN            ; Return from subroutine call
```

DG-03526

Figure 5.3 Subroutine code for an XJSR call

Figures 5.4 and 5.5 illustrate the result of executing the assembler code in Figure 5.3. For the XJSR instruction, the processor stores the return address into AC3 and jumps to the subroutine. With WSSVS as the first instruction of the subroutine, the processor stores PSR, AC0-AC2, old WFP, and carry (C) and AC3 (return address) onto the wide stack.

Although Figures 5.4 and 5.5 illustrate that the wide stack resides between 256_{10} and 355_{10}, the wide stack can be of any size and can reside anywhere within the segment.

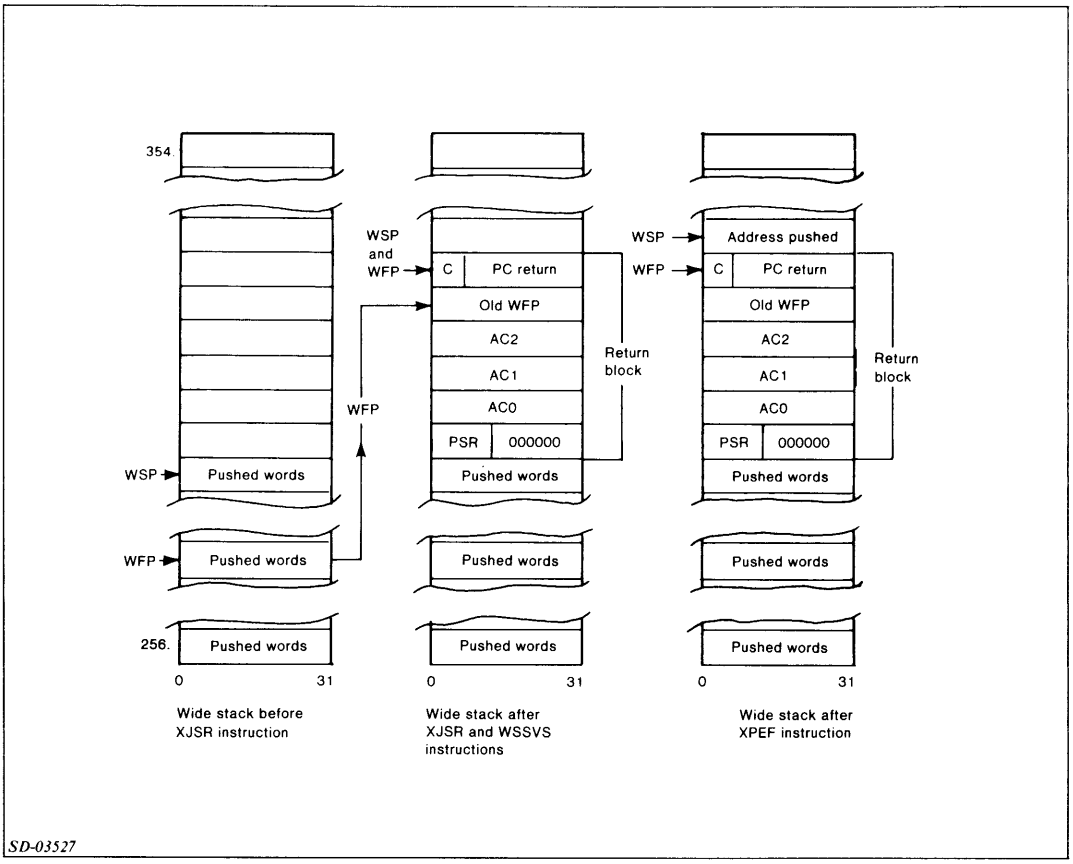


Figure 5.4 Wide stack operations from XJSR and WSSVS instructions

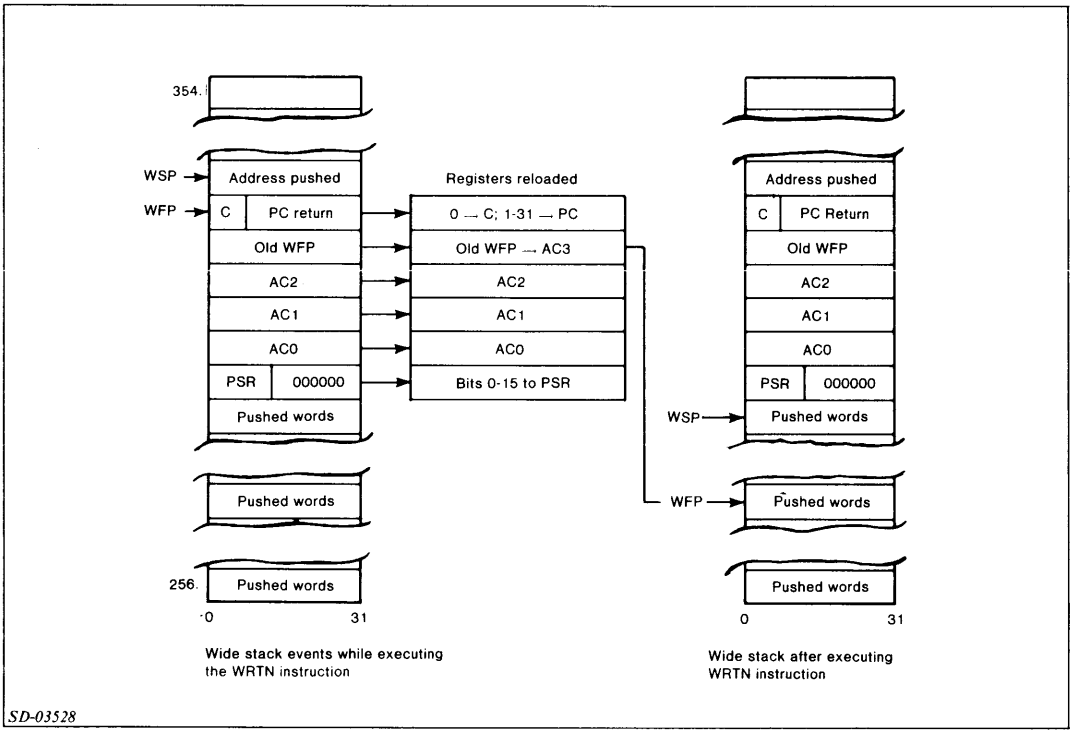


Figure 5.5 Wide stack operations from WRTN instruction

Transferring Program Control to Another Segment

The instructions listed in Table 5.6 transfer program control to or from another segment.

Instruction	Operation
LCALL	Call subroutine
WPOPB	Wide pop block
WRTN	Wide return
XCALL	Call subroutine
WRSTR	Wide restore from an I/O interrupt

Table 5.6 Segment transfer instructions

The LCALL and XCALL instructions initiate the transfer to another segment. The WRTN instruction returns program control from the LCALL and XCALL instructions. The WRSTR instruction returns program control from a base level I/O interrupt. The WPOPB instruction returns program control from an intermediate-level I/O interrupt. Refer to the Device Management chapter for a description of I/O interrupts.

The processor checks the direction of a transfer. A subroutine call must be inward (towards segment 0) and a return (from a subroutine call or I/O interrupt) must be outward (towards segment 7).

NOTE: *No segment crossing occurs with an interrupt request when the current segment equals zero and the interrupt-servicing code resides in segment 0.*

If the processor detects an invalid segment crossing, it does not execute the instruction; instead, it initiates a protection fault in the source segment. The processor sets AC1 to 7 for an illegal outward subroutine call, or sets AC1 to 8 for an illegal inward return.

NOTE: *The processor performs, without software assistance, all the functions necessary for a segment crossing.*

Subroutine Call

To transfer program control to another segment with the XCALL or LCALL instruction, the processor

1. Verifies that the instruction can access the destination segment.
2. Validates the entry point through a gate array in the destination segment.
3. Redefines the wide stack and transfers the call arguments to it.
4. Transfers program control.

Gate Array

A *gate array* is a series of locations that specify entry points (or gates) to the segment. The processor accesses a gate array through an indirect pointer in page zero of the destination segment. Figure 5.6 shows the format of a gate array.

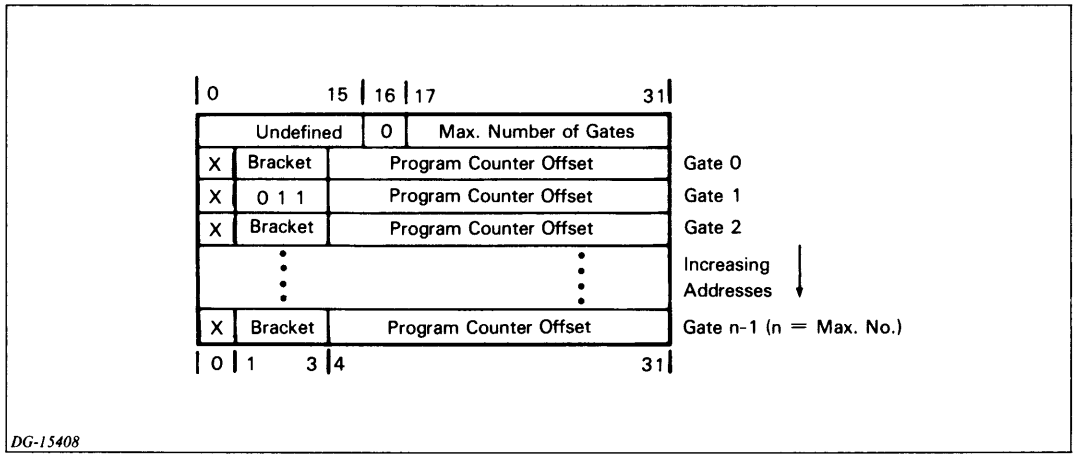


Figure 5.6 Gate array format

where

Undefined	The undefined bits mean that the processor does not care.
Maximum Number of Gates	The maximum number of gates specifies the total number of gates. If the maximum number is zero, the destination segment cannot be the target of an inward segment crossing.
x	The x bit means that the processor does not care.
Bracket	The bracket is the gate bracket. The gate bracket can be an unsigned integer in the range of zero to seven. The bracket identifies the highest source segment that can use the gate. For instance, if the Gate 1 gate bracket contains 011_2, only segments 0 through 3 can access the segment.
Program Counter Offset	The program counter offset is the address of the first instruction of the subroutine in the destination segment (target address).

Transfer

The processor interprets the effective address of the XCALL or LCALL instruction as shown in Figure 5.7.

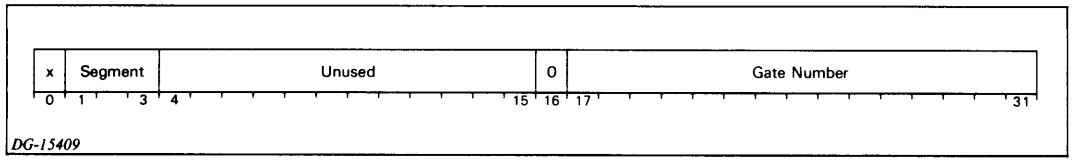


Figure 5.7 XCALL or LCALL effective address

where

x	The x bit (bit 0) is ignored by the processor.
Segment	The segment bits (bits 1-3) specify the segment number of the destination segment.
Unused	The unused bits (bits 4-15) are ignored by the processor.
Gate Number	The gate number (bits 17-31) specifies a gate in the destination segment. The processor uses the gate number as an index to an element (a gate) in the vectored array.

To perform a valid inward segment crossing, the processor

1. Tests for a valid segment by checking the validity bit in the segment base register.

   With an accessible segment, the processor continues with the next step. With an unaccessible segment, the processor aborts the call, sets AC1 to 3, and services the protection fault.

2. Checks for a valid gate by

   a. Comparing the gate number to the maximum number of gates.

      If the gate number is less than the maximum number of gates, the segment crossing continues.

   b. Comparing the segment number to the gate bracket number of the indexed gate.

      The segment number used for comparison is either the current segment of execution (if there was no indirection) or the segment of the last indirect address.

      If the segment number is equal to or less than the value in the gate bracket, the processor copies the segment number from the effective address to bits 1-3 of the program counter. Next, the processor copies the program counter offset (bits 4-31 from the indexed gate) to the program counter bits 4-31, and continues with the next step.

      If a gate number or a gate bracket comparison fails, the processor aborts the call, sets AC1 to 6, and services the protection fault. The protection fault occurs in the source segment.

3. Stores the wide frame pointer and wide stack pointer registers into page zero locations of the source segment.

   The values of the wide stack limit and wide stack base registers should be identical to the values in reserved memory.

4. Redefines the wide stack for the destination segment by loading the wide stack pointer, wide stack limit, and wide stack base registers from page zero locations of the destination segment.

   The WSAVS or WSAVR instruction will subsequently initialize the wide frame pointer.

5. Checks for a potential destination stack overflow.

   A parameter of the LCALL or XCALL instruction specifies the number of arguments to copy. The processor uses the parameter to determine if the number of arguments to copy exceeds the size of the wide stack.

   If the processor detects a potential overflow, it does not copy the arguments. It sets AC1 to 2, and processes a stack fault in the destination segment. The program counter word in the return block contains the address of the first instruction to execute in the destination segment.

   NOTE: *Refer to the Instruction Dictionary for the LCALL and XCALL argument count description.*

6. Copies the arguments from the source stack to the destination stack, if no potential overflow exists.

   The order of the arguments in the destination stack matches the order of the arguments in the source stack.

   NOTE: *The copying of arguments is interruptable.*

7. Pushes a double word that contains the processor status register and the number of arguments pushed.

8. Executes the first instruction of the subroutine.

   A wide save instruction (WSAVR or WSAVS) must be the first instruction of the subroutine. Either instruction would push a return block onto the destination wide stack and load the wide frame pointer with the updated value of the wide stack pointer.

Trojan Horse Pointers

When executing a subroutine in another segment, the processor uses the access privileges of the destination segment to determine the validity of the reference. A *trojan horse pointer* exists if one of the arguments passed from the source segment points to a location in the destination segment. (A privileged access fault would occur if a program refers to a location in a lower numbered segment.)

For example, a trojan horse pointer can exist when a program in segment 6 calls a subroutine in segment 2, and one of the arguments passed is a pointer to information in segment 2.

You can protect against a trojan horse pointer by using the validate word pointer (VWP) or validate byte pointer (VBP) instruction to ensure that the source segment and destination segment are identical.

The processor protects against a trojan horse pointer when it executes a character move instruction that moves data in descending order (such as WCMT and WCMV). The processor checks each data transfer and ensures that the source segment and destination segment remain the same.

Subroutine Return

As the last instruction of the subroutine, use the wide return instruction (WRTN) to return program control from the LCALL or XCALL. The processor places the contents of the wide frame pointer into the wide stack pointer. Then, the processor

1. Pops the six double word return block.

   The processor pushed the first five double words of the return block when it executed the WSAVR or WSAVS instruction. The processor pushed the sixth double word (processor status register and the number of arguments) when it executed the LCALL or XCALL instruction.

2. Stores the wide frame pointer and wide stack pointer registers into page zero locations of the source segment.

   The values of the wide stack limit and wide stack base registers should be identical to the values in reserved memory.

3. Redefines the wide stack for the destination segment by loading the wide stack limit, wide stack base, and wide frame pointer registers from page zero locations of the destination segment.

4. Calculates the address of the double word that precedes the arguments of the calling sequence and loads the wide stack pointer with the double word.

5. Executes the instruction after the LCALL or XCALL instruction.

   As a result of step 1, the processor loads the program counter with the return address in the destination segment.

Fault Handling

While executing an instruction, the processor performs certain checks on the operation and the data. If the processor detects an error, a privileged or nonprivileged fault occurs before executing the next instruction. Table 5.7 lists the faults.

Fault	Type
Nonresident page	Privileged
Protection violation	Nonprivileged
Stack operation	Nonprivileged
Fixed-point computation	Nonprivileged
Floating-point computation	Nonprivileged
Invalid decimal or ASCII data format	Nonprivileged

Table 5.7 Faults

When the processor detects a fault, it pushes a return block onto the stack and jumps to the fault handler through the indirect pointer in reserved memory. The initial and indirect pointers to a fault handler (except to a page fault handler) are 16 bits. Levels of indirection, if any, occur within the segment initially containing the pointer. A nonprivileged fault pointer is located in page zero of the current segment. A privileged fault pointer is located in page zero of segment 0.

If a privileged fault occurs while handling a nonprivileged fault, the processor aborts the nonprivileged fault and processes the privileged fault. Refer to the Memory and System Management chapter for privileged fault handling.

If an I/O interrupt occurs after detecting the nonprivileged fault, the processor pushes the fault return block, updates the program counter to the first instruction of the fault handler, and then services the I/O interrupt. Upon returning from the I/O interrupt, the processor services the nonprivileged fault.

To service a nonprivileged fault, the processor

1. Sets AC1 to a value that identifies the fault when a stack fault or a decimal/ASCII fault occurs.

   Appendix D lists the fault codes.

2. Pushes a fault return block onto the stack.

   The fault return block contains the address of the instruction that the processor was executing at the time of the fault.

3. Checks for stack overflow.

   If a stack overflow occurs, the processor pushes a stack fault return block onto the stack and processes the stack fault. The stack fault return block contains the return address to the original fault.

   If no stack overflow occurs, the processor continues to service the original fault.

4. Jumps to the fault handler.

   The last instruction of a wide fault handler should be a WPOPB instruction for the processor to continue executing the interrupted program.

Protection Violations

The processor detects a protection violation for an invalid memory reference, invalid I/O operation, or illegal instruction (such as a privileged instruction or an unimplemented opcode). The Unimplemented Instructions section describes unimplemented opcodes; the Memory and System Management chapter describes some of the fault conditions listed in Table 5.8.

Since an operation could produce multiple protection violations, the processor imposes priorities on the faults. When two or more faults occur simultaneously, the processor services the highest priority fault and ignores lower priority faults. Table 5.8 lists the protection violation faults in the order of priority. For instance, if writing to a write protected page in an inner ring, the processor services the inward ring reference protection violation (with priority 2) and ignores the write protection violation (with priority 4).

Level of Priority	Fault Description
0	Privileged or I/O instruction violation
1	Indirect addressing violation
2	Inward reference violation
3	Segment validity violation
4	Page table validity violation
5	Read, write, or execute access violation
6	Segment crossing violation
7	Unimplemented opcode or instruction

Table 5.8 Priority of protection violation faults

When the processor detects a protection violation (see Figure 5.8), it checks the contents of reserved memory location 36_8 in the current ring for the protection fault handler. The contents of this location contain a 16-bit non-indirectable pointer which determines in which segment the protection fault will be serviced.

~ If this location is zero, no outer ring protection fault handler has been defined, and the processor performs the following:

1. Stores the contents of the wide stack pointer and the wide frame pointer into the page zero locations of the current segment. (The values of the stack limit and stack base registers should be identical to the values in reserved memory.)
2. Crosses to segment 0 (if the current segment is 1 to 7).
3. Redefines the wide stack for segment 0. The processor initializes the wide stack pointer, wide stack limit, and wide stack base registers from segment 0, page zero locations.
4. Pushes a fault return block, as shown in Table 5.9, onto the segment 0 stack.

   NOTE: *If a protection violation occurs while attempting to get to the segment 0 protection fault handler (while pushing the fault return block), an infinite protection fault is generated and the system halts.*

~ If this location is non-zero, the current segment has a protection fault handler defined, and the processor performs the following:

1. Pushes a fault return block, as shown in Table 5.9, onto the current segment's stack.

   NOTE: *If a protection violation occurs while attempting to get to the outer ring protection fault handler, control is transferred to the segment 0 protection fault handler. (The only way to generate a protection fault while getting to the outer ring protection fault handler is to get a protection fault while pushing the return block onto the stack.)*
2. Sets the PSR to zero.
3. Initializes AC0, AC1, and AC2.

   Sets AC0 equal to the address of the instruction (offending PC) causing the fault.

   Sets AC1 equal to a value identifying the fault. Table 5.10 lists the protection fault codes.

   Sets AC2 equal to the specific address (offending address) that caused the reference problem, if applicable (bit 0 is undefined). See Table 5.10.
4. Checks for stack overflow.

   If a stack overflow occurs, the processor pushes a stack fault return block onto the stack and processes the stack fault. The stack fault return block contains the return address to the protection fault.

   If no stack overflow occurs, the processor continues to service the protection fault.

5. Sets the PSR to zero.

6. Initializes AC0, AC1, and AC2.

   Sets AC0 equal to the address of the instruction (offending PC) causing the fault.

   Sets AC1 equal to a value identifying the fault. Table 5.10 lists the protection fault codes.

   Sets AC2 equal to the specific address (offending address) that caused the reference problem, if applicable (bit 0 is undefined). See Table 5.10.

7. Checks for stack overflow.

   If a stack overflow occurs, the processor pushes a stack fault return block onto the stack and processes the stack fault. The stack fault return block contains the return address to the protection fault.

   If no stack overflow occurs, the processor continues to service the protection fault.

8. Jumps to the fault handler and executes the first instruction. Segment 0 reserved memory location 36_8 contains the 16-bit non-indirect starting address of the protection violation fault handler.

5. Jumps to the fault handler and executes the first instruction. The current segment's reserved memory location 36_8 contains the 16-bit non-indirect starting address of the protection violation fault handler.

NOTE: *If a protection violation occurs during execution of the outer ring protection fault handler routine, the same outer ring protection fault routine will handle the new protection fault. Unless a protection violation is generated while pushing a subsequent return block onto the stack, an infinite loop is created. Note that an infinite loop is not an infinite protection fault; it does not halt the machine.*

One method of avoiding the above situation is to save reserved memory location 36_8, write zeroes to the location, and then restore the location to the previously-saved value before returning from the protection fault handler.

Word Number in Block Pushed	Contents
1	PSR bits 0-15 of the processor status register
2	bits 0-15 equal 16 0's
3-4	AC0 bits 0-31
5-6	AC1 bits 0-31
7-8	AC2 bits 0-31
9-10	AC3 bits 0-31
11-12	Bit 0 equals the CARRY flag Bits 1-31 equal the PC of execution if fault type is privileged or I/O, else it is undefined.

Table 5.9 Fault return block

During the servicing of a protection violation:

- If an I/O interrupt request occurs, the processor executes the first instruction of the protection violation fault handler before servicing the interrupt request.
- If a protection violation fault occurs while handling any other type of nonprivileged fault, the processor aborts the first fault and processes the protection volation fault. The return block pushed onto the stack for the protection violation fault is undefined, as are the contents of AC0 and AC1.

Code (octal)	AC 2 Address	Meaning	Explanation
0	Y	Read violation	Bit 2 of the specified PTE contains a zero
1	Y	Write violation	Bit 3 of the specified PTE contains a zero
2	Y	Execute violation	Bit 4 of the specified PTE contains a zero
3	Y	Validity violation (SBR or PTE)	Bit 0 of the specified SBR or PTE contains a zero
4	Y	Inward address reference	Attempted data access to a location in an inner segment
5	Y	Defer (indirect) violation	More than 15 levels of indirection specified
6	N	Illegal gate	Gate number specified in an inward call is greater than or equal to the maximum number of gates; or a gate bracket access violation
7	Y	Outward call	Attempted transfer of control from the current segment to another segment with an outward subroutine call
10	Y	Inward return	Attempted transfer of control from the current segment to another segment with an inward return from a subroutine
11	N	Privileged instruction violation	Attempted use of a privileged instruction in a segment other than segment 0
12	N	I/O protection violation	Attempted use of an I/O instruction when bit 3 of the current segment's SBR is set to zero
14	N	Invalid microinterrupt return block	The return block created during a microinterrupt is incorrect.
15	N	Unimplemented instruction	The specified instruction opcode is not implemented on this machine.

Table 5.10 Protection fault codes

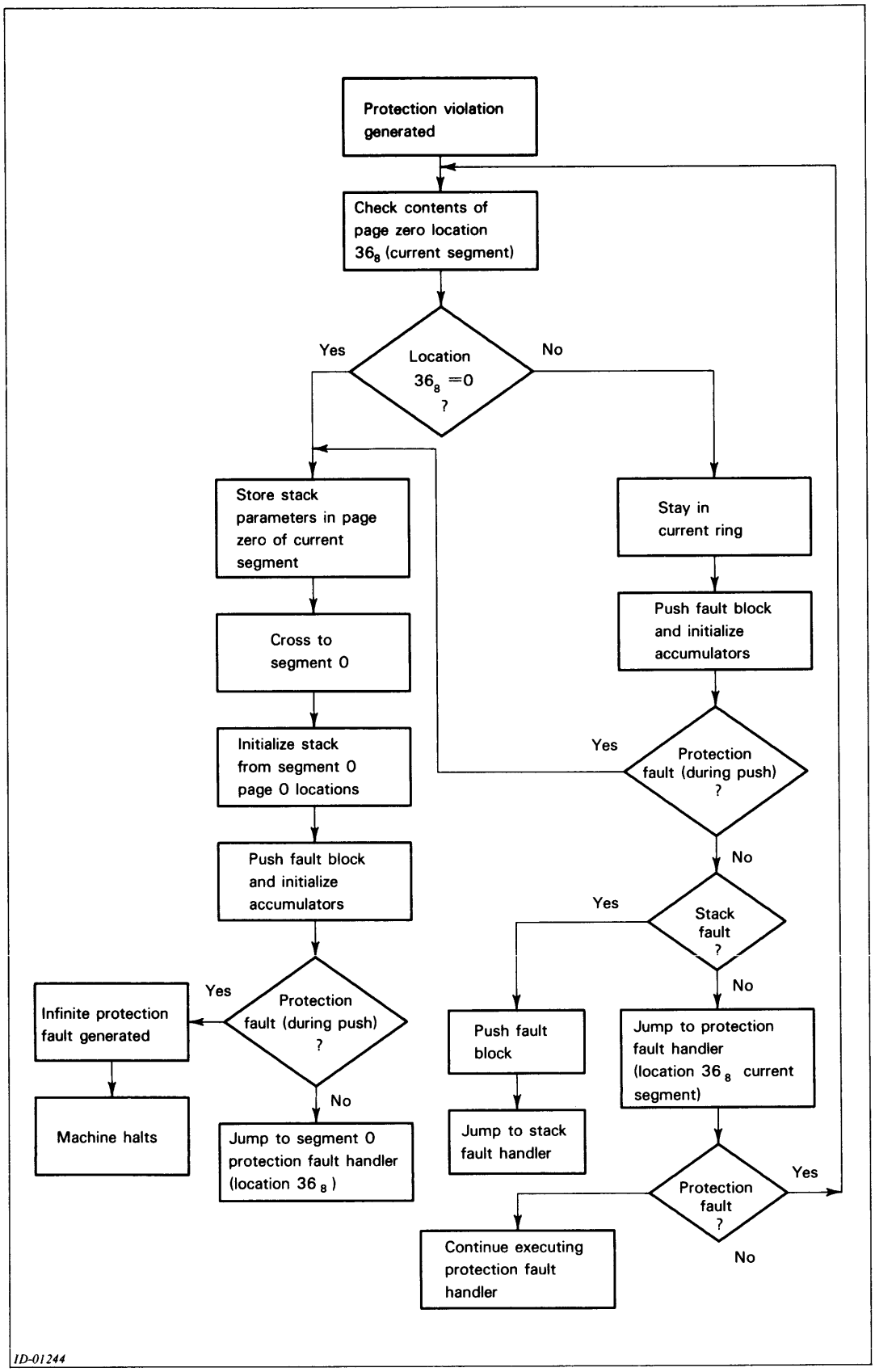


Figure 5.8 Priority Violation Sequence

Unimplemented Instructions

The processor checks for a valid instruction opcode before executing an instruction. If the instruction is implemented on your machine, the operation is performed by hardware.

If the instruction is not implemented or the opcode is invalid, the processor takes a protection fault and reports the fault code for an unimplemented instruction.

NOTE: *Since an unimplemented instruction is not dependent on the address translator, the instruction can cause a protection fault when the machine is in physical mode. In this case, the processor uses the protection fault handler specified in physical location* 36_8.

Fixed-Point Overflow Fault

The processor detects a fixed-point overflow when attempting division by zero or when calculating a two's complement number that is too large to store in memory or in a fixed-point accumulator. The processor sets the overflow flag (OVR) to one.

For the processor to service the fixed-point fault (or trap), you must set the overflow fault mask (OVK) to one before the processor sets the overflow flag. Use the FXTE instruction to set OVK to one, and the FXTD instruction to set OVK to zero.

If the OVK mask equals zero when the processor sets the OVR flag to one, the processor ignores the overflow. However, OVR remains set to one until explicitly changed. The processor continues normal program execution with the next sequential instruction.

If the OVK fault mask equals one, the processor initiates a fixed-point overflow fault at the end of the current fixed-point instruction. The processor sets AC1, pushes a wide return block, and jumps to the fault handler through the 16-bit indirect pointer in reserved memory. Table 5.11 shows the fixed-point fault return block.

Word Number in Block Pushed	Contents
1	PSR bits 0-15 of the processor status register
2	bits 0-15 equal 16 0's
3-4	AC0 bits 0-31
5-6	AC1 bits 0-31
7-8	AC2 bits 0-31
9-10	AC3 bits 0-31
11-12	PC bit 0 equals the CARRY flag bits 1-31 address of the instruction following the fault-causing instruction

Table 5.11 Fixed-point fault return block

NOTE: *Although some instructions initialize OVK and OVR at the same time, no overflow fault occurs.*

The PSR word in the return block contains OVR set to zero; OVK set to one; and IRES unchanged. The return address is the address of the instruction the processor executes after servicing the fault.

After the push, AC0 contains the address of the instruction that caused the fault. The processor sets the processor status register to zero and jumps to the fault handler through the 16-bit indirect pointer in reserved memory.

Floating-Point Faults

The processor detects a floating-point error, when attempting division by zero, executing an IIS instruction with an invalid input argument, or when calculating a number that is too large to store in memory or in a floating-point accumulator. The processor sets both the appropriate FPSR fault flag (OVF, UNF, INV, and MOF) and the ANY flag to one. If the error is an invalid input argument, the processor also returns a code to the FPSR INP bits.

For the processor to service a floating-point fault (or trap), you must set the floating-point fault mask (TE) to one before the processor sets a floating-point fault flag. Use the FTE instruction to set TE to one and the FTD instruction to set TE to zero.

If the TE fault mask equals zero when the processor sets a floating-point fault flag to one, the processor ignores the overflow. The processor continues normal program execution with the next sequential instruction.

If the TE fault mask equals one, when the processor sets a floating-point fault flag to one, the processor initiates a floating-point overflow fault at the end of the current floating-point instruction. The processor jumps to the fault handler through the 16-bit indirect pointer in reserved memory.

The processor services narrow and wide floating-point faults using the same pointer. If the first word (instruction) of the fault handler contains bit 0 set to one and bits 12 through 15 set to 1001_2, the processor pushes a wide return block onto the wide stack. Otherwise, the processor pushes a narrow return block onto the narrow stack. Table 5.12 shows the wide return block and Table 5.13 shows the narrow return block.

Word Number in Block Pushed	Contents
1	PSR bits 0-15 of the processor status register
2	bits 0-15 equal 16 0's
3-4	AC0 bits 0-31
5-6	AC1 bits 0-31
7-8	AC2 bits 0-31
9-10	AC3 bits 0-31
11-12	PC bit 0 equals the CARRY flag bits 1-31 address of the instruction following the fault-causing instruction

Table 5.12 Wide floating-point fault return block

Word Number in Block Pushed	Contents*
1	AC0 bits 16-31
2	AC1 bits 16-31
3	AC2 bits 16-31
4	AC3 bits 16-31
5	PC bits 17-31 address of instruction following the fault-causing instruction

Table 5.13 Narrow floating-point fault return block

*Bits 1-15 of a word correspond to bits 17-31 of a register.

The return address in the return block is the address of the next instruction that the processor executes after servicing the fault. Use the store floating-point status instruction (LFSST or FSST) to determine the address of the floating-point instruction that caused the fault.

After the pushing the return block, the processor

1. Sets the processor status register to zero (for a wide floating point fault).
2. Sets the TE fault mask to zero.
3. Transfers program control to the floating-point fault handler.

Decimal and ASCII Data Faults

The processor checks for a valid decimal or ASCII data type and for valid data when executing an edit or a load/store integer instruction. If either the data type or the data is invalid, the fault occurs at the end of the current instruction.

The processor pushes a wide return block onto the wide stack if executing a 32-bit instruction (such as WEDIT or WSTIX). The processor pushes a narrow return block onto the narrow stack if executing a 16-bit instruction (such as ECLIPSE C/350 EDIT or STIX).

The length and width of the return block depends on the fault that occurs and the instruction that causes it. For example, the WEDIT instruction uses the wide stack for temporary storage. When a fault occurs, the processor pushes the return block in addition to the temporary words that the WEDIT instruction requires.

After pushing the return block, the processor sets the processor status register to zero and places the fault code in AC1 bits 16-31. AC0 contains the value of the program counter for the instruction that caused the fault.

Program control jumps to the fault handler through the 16-bit indirect pointer in reserved memory. Both the wide and narrow faults use the same fault pointer and handler.

Table 5.14 lists the decimal and ASCII fault codes. The first and second columns list the code that appears in AC1. The third column lists the type of return block pushed. The fourth column lists the instruction that caused the fault. The last column describes the conditions that can cause the fault.

Code Returned in AC1		Return Block Type	Faulting Instruction	Meaning
Narrow	Wide			
000000	100000	2	EDIT, WEDIT	An invalid digit or alphabetic character encountered during execution of one of the following subopcodes: DMVA, DMVF, DMVN, DMVO, DMVS
000001	100001	1	LDIX, STIX	Invalid data type (7)
		3	EDIT, WEDIT WLDIX, WSTIX WDMOV, WDCMP, WDINC, WDDEC	Invalid data type (6 or 7)
000002	100002	2	EDIT, WEDIT	DMVA or DMVC subopcode with source data type 5; AC2 contains the data size and precision
000003	100003	2	EDIT, WEDIT	An invalid opcode; AC2 contains the data size and precision
000004	100004	1	STI, LDI, WSTI, WLDI	Number too large to convert to specified data type . number $> (10^{16}) - 1$
			STIX, LDIX, WSTIX, WLDIX	Number too large to convert to specified data type. Number $> (10^{32}) - 1$
000006	100006	1	WLSN, WLDI, LSN, LDI LDIX, WLDIX	Sign code is invalid for this data type
		3	EDIT, WEDIT WDMOV, WDCMP, WDINC, WDDEC	
000007	100007	1	WLSN, WLDI, WLDIX, LSN LDI, LDIX	Invalid digit
		3	WDMOV, WDCMP, WDINC, WDDEC	

Table 5.14 Decimal and ASCII fault codes

Wide Fault Return Blocks

Tables 5.15 through 5.17 list the contents and types of wide return blocks. After the processor pushes a wide return block, the accumulators retain their original contents, except that AC1 contains the fault code.

Word Number in Block Pushed	Contents
1	PSR bits 0-15 of the processor status register
2	bits 0-15 equal 16 0's
3-4	AC0 bits 0-31 unchanged
5-6	AC1 bits 0-31 original descriptor
7-8	AC2 bits 0-31 original source indicator (destination indicator for WSTI or STIX instruction)
9-10	AC3 bits 0-31 undefined
11-12	PC bit 0 equals the CARRY flag
	bits 1-31 of the decimal instruction address causing the fault

Table 5.15 Wide return block for decimal data (type 1) fault

Word Number in Block Pushed	Contents
1	PSR bits 0-15 of the processor status register
2	bits 0-15 equal 16 0's
3-4	AC0 bits 0-31 current value of P (byte pointer to subopcode that caused the fault)
5-6	AC1 bits 0-31 original descriptor
7-8	AC2 bits 0-31 undefined
9-10	AC3 bits 0-31 undefined
11-12	PC bit 0 equals the CARRY flag
	bits 1-31 of the WEDIT instruction address causing fault

Table 5.16 Wide return block for ASCII data (type 2) fault

Word Number in Block Pushed	Contents
1	PSR bits 0-15 of the processor status register
2	bits 0-15 equal 16 0's
3-4	AC0 bits 0-31 (original source descriptor for WDMOV and WDCMP)
5-6	AC1 bits 0-31 original descriptor
7-8	AC2 bits 0-31 (original source pointer for WDMOV and WDCMP)
9-10	AC3 bits 0-31 (original destination pointer for WDMOV, WDCMP, WDINC, and WDDEC)
11-12	PC bit 0 equals the CARRY flag bits 1-31 of the instruction address causing the fault

Table 5.17 Wide return block for ASCII data (type 3) fault

Narrow Fault Return Blocks

Tables 5.18 through 5.20 list the contents and types of narrow return blocks. After the processor pushes a narrow return block, the accumulators retain their original contents, except that AC1 contains the fault code.

Word Number in Block Pushed	Contents
1	AC0 bits 16-31 unchanged
2	AC1 bits 16-31 contain original descriptor
3	AC2 bits 16-31 contain original source indicator (destination indicator for WSTI or STIX)
4	AC3 bits 16-31 undefined
5	PC bits 17-31 of the decimal instruction address causing the fault

Table 5.18 Narrow return block for decimal data (type 1) fault

Word Number in Block Pushed	Contents
1-4	reserved (for ECLIPSE compatibility)
5	AC0 bits 16-31 contain current value of P (byte pointer to subopcode that causes the fault)
6	AC1 bits 16-31 original descriptor
7	AC2 bits 16-31 undefined
8	AC3 bits 16-31 undefined
9	PC bits 17-31 of the decimal instruction address causing the fault

Table 5.19 Narrow return block for ASCII data (type 2) fault

Word Number in Block Pushed	Contents
1	AC0 bits 16-31 unchanged
2	AC1 bits 16-31 original descriptor
3	AC2 bits 16-31 undefined
4	AC3 bits 16-31 undefined
5	PC bits 17-31 of the decimal instruction address causing the fault

Table 5.20 Narrow return block for ASCII data (type 3) fault

Stack Faults

The processor checks for a narrow stack fault after a narrow stack operation, and checks for a wide stack fault after a wide stack operation. When a stack overflow occurs, the program overwrites the data in the area beyond the stack. When a stack underflow occurs, the program accesses incorrect information. Once detected, the processor always services the narrow or wide stack fault.

The narrow stack is a series of single words managed by three reserved memory words. The narrow stack supports program development and upward program compatibility for 16-bit programs (such as the ECLIPSE C/350).

Refer to the ECLIPSE C/350 Programming chapter and the ECLIPSE C/350 Principles of Operation manual for additional information on narrow stack operations.

Wide Stack Fault Operations

After a wide push operation, the processor compares the contents of the wide stack pointer to the contents of the wide stack limit. If the wide stack pointer value is greater than the wide stack limit value, the processor detects a wide stack overflow fault.

After a wide pop operation, the processor compares the contents of the wide stack pointer to the contents of the wide stack base. If the wide stack pointer value is less than the wide stack base value, the processor detects a wide stack underflow fault.

You can disable wide stack overflow fault detection by loading the value 37777777777_8 into the wide stack limit register. You can disable wide stack underflow fault detection by loading the value 20000000000_8 into the wide stack base register.

When a wide stack fault occurs, the processor

1. For a wide stack underflow, sets the wide stack pointer equal to the wide stack limit.
2. Pushes a wide return block onto the wide stack (see Table 5.21).

   The return address word in the wide return block points to the next instruction that the processor executes after servicing the fault.
3. Sets the OVK, OVR, and IRES flags (PSR flags) to zero.
4. Sets bit 0 of the wide stack pointer to zero.
5. Sets bit 0 of the wide stack limit to one.
6. Updates the wide stack pointer and reserved memory locations in the current segment.
7. Loads AC0 with the address of the instruction that caused the fault.
8. Loads AC1 with the code that describes the fault (see Table 5.22).
9. Jumps to the wide stack fault handler through the 16-bit indirect pointer in page zero of the current segment.

   If an I/O interrupt occurs before the processor executes the first instruction of the fault handler, the program counter word in the return block points to the first instruction of the fault handler. Thus, an I/O interrupt waits until the processor pushes the return block and updates the program counter.

Word Number in Block Pushed	Contents
1	PSR bits 0-15 of the processor status register
2	bits 0-15 equal 16 0's
3-4	AC0 bits 0-31
5-6	AC1 bits 0-31
7-8	AC2 bits 0-31
9-10	AC3 bits 0-31
11-12	PC bit 0 equals the CARRY flag bits 1-31 of the instruction address causing the fault if a type 1 fault; else, the instruction address of the next executing instruction.

Table 5.21 Wide stack fault return block

AC1 Code	Meaning
000000	Overflow on every stack operation other than SAVE, WMSP, or segment crossing
000001	Underflow or overflow would occur if the instruction were executed -- WMSP, WSSVR, WSSVS, WSAVR, WSAVS (PC in return block refers to the instruction that caused the stack fault.)
000002	Too many arguments on a cross segment call
000003	Stack underflow
000004	Overflow due to a return block pushed as a result of a microinterrupt or fault

Table 5.22 Wide stack fault codes

If you determine that you must write a wide stack fault handler, the handler must

1. Determine the nature of the fault (underflow or overflow).
2. Reset bit 0 of the wide stack pointer and the wide stack limit to the original values.
3. Take any other appropriate action, such as allocating more stack space or terminating the program.
4. Use a WPOPB instruction as the last instruction of the fault handler.

Narrow Stack Fault Operations

After a narrow push operation, the processor compares the contents of the narrow stack pointer to the contents of the narrow stack limit. If the stack pointer value is greater than the stack limit value, the processor detects a narrow stack overflow fault.

After a narrow pop operation, the processor compares the contents of the narrow stack pointer to 401_8. If the stack pointer value is less than 400_8 and bit 0 of the narrow stack limit is zero, the processor detects a narrow stack underflow fault.

You can disable narrow stack overflow fault detection by setting bit 0 of the narrow stack pointer to zero, and bit 0 of the stack limit to one. You can disable narrow stack underflow fault detection by

- Starting the narrow stack at a location greater than 401_8.

  If the narrow stack starts at location greater than 401_8, underflow still occurs when the value of the stack pointer becomes less than 400_8. The processor can detect underflow if a program pops enough words from the narrow stack to cause the narrow stack pointer to wraparound.

- Setting bit 0 of either the narrow stack pointer or the narrow stack limit to one.

  If bit 0 of the narrow stack pointer or narrow stack limit is set to one, either all or part of the stack may reside in page zero, or the stack may underflow onto page zero without interference from the narrow stack fault handler.

When a narrow stack fault occurs, the processor

1. Sets the narrow stack pointer equal to the narrow stack limit if the narrow stack underflow occurs.
2. Sets bit 0 of the narrow stack pointer to zero and bit 0 of the narrow stack limit to one.

   Thus, the narrow stack limit is (temporarily) larger than the narrow stack pointer, which disables overflow fault detection.
3. Pushes a narrow return block onto the narrow stack (see Table 5.23).

   The return address word in the narrow return block points to the next instruction the processor executes after servicing the fault.
4. Jumps to the narrow stack fault handler through the 16-bit indirect pointer in page zero of the current segment.

   If an I/O interrupt occurs before the processor executes the first instruction of the fault handler, the program counter word in the return block points to the first instruction of the fault handler. Thus, an I/O interrupt waits until the processor pushes the return block and updates the program counter.

Word Number in Block Pushed	Contents
1	AC0 bits 16-31
2	AC1 bits 16-31
3	AC2 bits 16-31
4	AC3 bits 16-31
5	PC bit 16 equals the CARRY flag bits 17-31 of the instruction address causing the fault

Table 5.23 Narrow stack fault return block

If you determine that you must write a narrow stack fault handler, the handler must

1. Determine the nature of the fault (underflow or overflow).
2. Reset bit 0 of the narrow stack pointer and the narrow stack limit to the original values.
3. Take any other appropriate action, such as allocating more stack space or terminating the program.
4. Use a POPB instruction as the last instruction of the fault handler.

Chapter 6
Queue Management

Queues

A *queue* is a variable-length list of linked entries that has a beginning and an end. The operating system uses queues to keep track of processes that it must run (ready queue), files that must be printed on the line printer, pages that are resident in physical memory, and so on.

An entry in a queue is called a *data element*. Adding a data element to a queue is called *enqueuing*. Removing a data element is called *dequeuing*. The ends of a queue are called the *head* and the *tail*. A typical first in, first out (FIFO) queue has data elements enqueued at the tail and dequeued at the head.

One of the advantages of using a queue rather than a single threaded list is that queue data elements refer to the data elements that precede *and* follow them. In other words, the processor queues use a priority-based structure. This means that data elements can be *enqueued* anywhere in the queue, not just at the tail. Conversely, data elements can be *dequeued* anywhere in the queue, not just at the head.

New entries are added to the queue when service (such as the name of a new file to be printed) is required, and they are removed from the queue after they are of no further use. A queue may be empty, it may have only one entry, or it may have many entries.

Building a Queue

For the data elements to be linked together, each data element must contain two addresses, called *links*. One of the links contains the effective word address of the following data element in the queue: the *forward link*. The other link contains the effective word address of the preceding data element in the queue: the *backward link*.

The forward and backward links do more than refer to the adjacent queue data elements. They also indicate the elements that are currently at the head and tail of the queue. If a data element's forward link contains -1, then that data element is at the tail of the queue. If a data element's backward link contains -1, then that data element is at the head of the queue. Note that a data element containing -1 in both its forward and backward links is the only data element currently in the queue.

A data element contains user information as well as the forward and backward links. This user information can either precede or follow the forward and backward links (see Tables 6.1 and 6.2). The user determines the structure and the meaning of the information.

Position in Data Element	Contents
First double word	Forward link
Second double word	Backward link
Next *n* double words	User information

Table 6.1 Data element with user data following links

Position in Data Element	Contents
First *n* double words	User information
(*n* + 1)th double word	Forward link
(*n* + 2)th double word	Backward link

Table 6.2 Data element with user data preceding links

Also, note that the length of the user information in the data elements can vary, since the links of each data element always refer to other links and not to user information. The search queue instructions, however, do refer to the user information, so make sure that any programs using these instructions take the length of the user information into account.

Queue Descriptor

Each queue uses a *queue descriptor* that indicates the current head and tail of the queue. A queue descriptor is two 32-bit words. The first double word contains the address of the data element that is currently at the head of the queue: the second contains the address of the data element that is currently at the tail of the queue (see Figure 6.1).

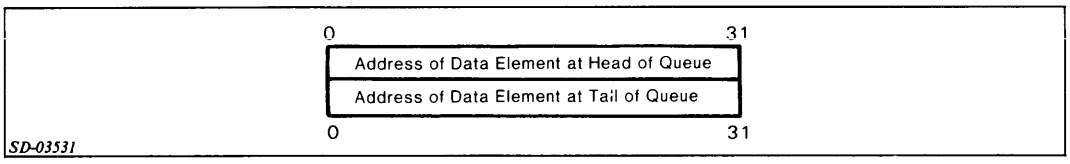


Figure 6.1 Format of queue descriptor

Setting Up and Modifying a Queue

To define an empty queue, create a queue descriptor that contains -1 in both of its pointers. To enqueue a data element into the empty queue, load the address of the data element into both double words of the queue descriptor (indicating a one-element queue) and load -1 into the data element's forward and backward links. To enqueue or dequeue a data element anywhere in the queue, specify the queue descriptor and the address of some data element in the queue. The descriptor and address specified act as reference points that the processor uses to enqueue the data element at the right point or to dequeue the appropriate data element.

Note that you can create a new one-element queue in one step. To produce a one-element queue, create a queue descriptor that contains the address of a data element in both double words. Then, load both of the links of the particular data element with -1.

Examples

The examples below demonstrate how you can form queues, how enqueuing and dequeuing works, and how the processor updates the various links and descriptors.

Queue Descriptor of an Empty Queue

Figure 6.2 shows the queue descriptor for an empty queue.

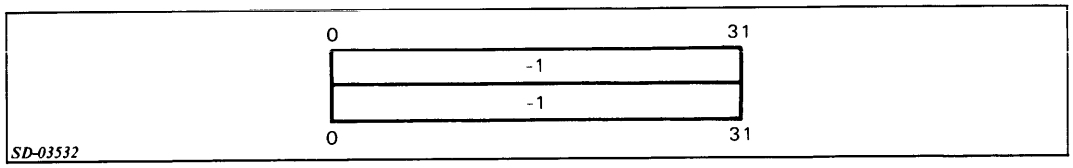


Figure 6.2 Queue descriptor for an empty queue

Enqueuing a Data Element into an Empty Queue

Figure 6.3 illustrates how the processor enqueues a data element (located at location A) into an empty queue. After the enqueue, the processor updates the queue descriptor. The descriptor shows that the queue has only one element, A. At location A, the first word of the data element contains the forward link -1. The last word contains the backward link of -1.

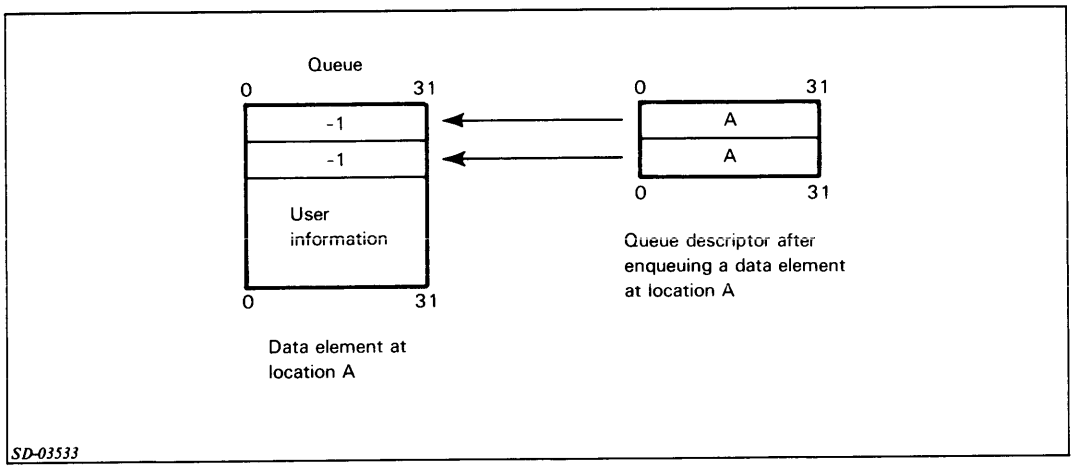


Figure 6.3 Data element enqueued into an empty queue

Enqueuing a Data Element at the Head of a Queue

Figure 6.4 illustrates how the processor enqueues a data element (located at location B) at the head of the queue before data element A. After the enqueue, the processor updates the queue descriptor to refer to the new head and tail. It also changes the backward link of data element A to refer to the preceding data element (B). The links of data element B show that it is the head of the queue and that element A follows it.

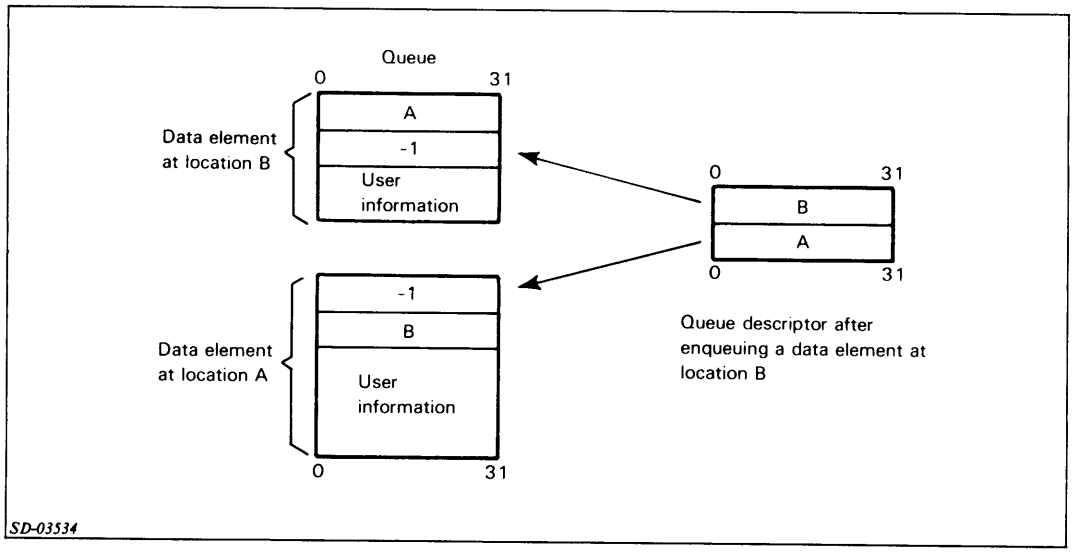


Figure 6.4 Data element enqueued at head of queue

Enqueuing a Data Element at the Tail of a Queue

Figure 6.5 illustrates how the processor enqueues a data element (located at location C) at the tail of the queue, after data element A. The -1 in data element B's backward link shows that B is the head of the queue. The -1 in data element C's forward link shows that C is the tail of the queue. The queue descriptor also indicates the new head and tail of the queue.

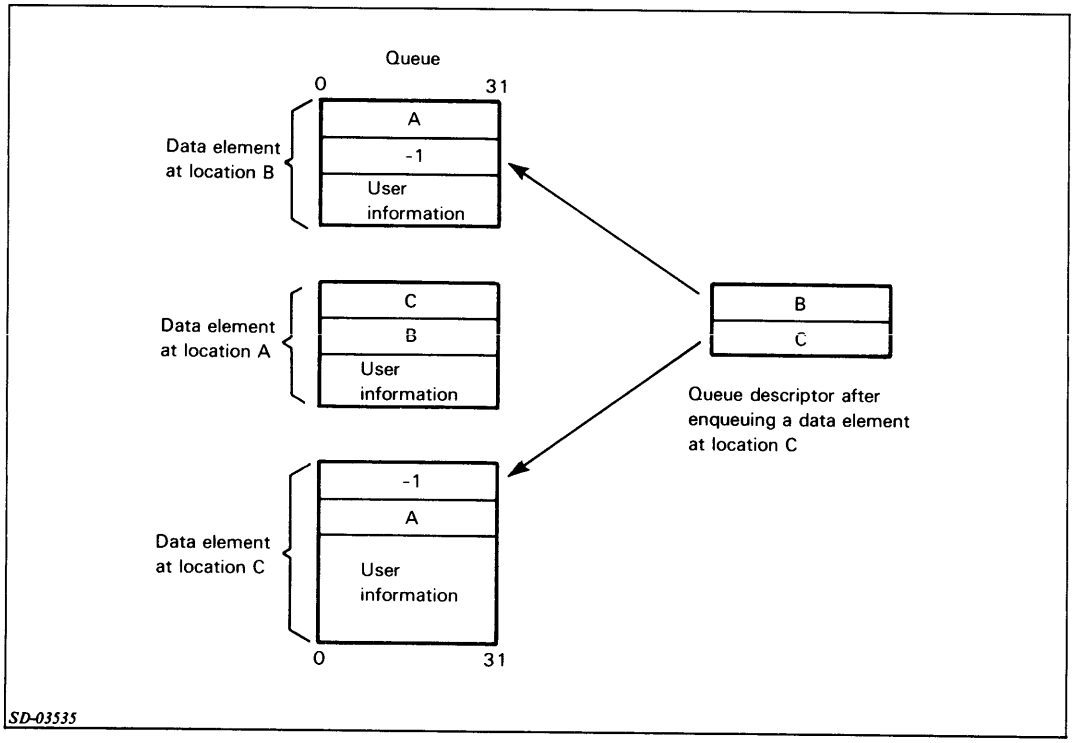


Figure 6.5 Data element enqueued at tail of queue

Dequeueing a Data Element

Figure 6.6 illustrates how the processor dequeues data element B. After the dequeue, the processor updates the queue descriptor to show the new head (A). A's backward link shows that it is the new head. C's links remain unchanged, since C is still the tail of the queue, and A is still the following data entry.

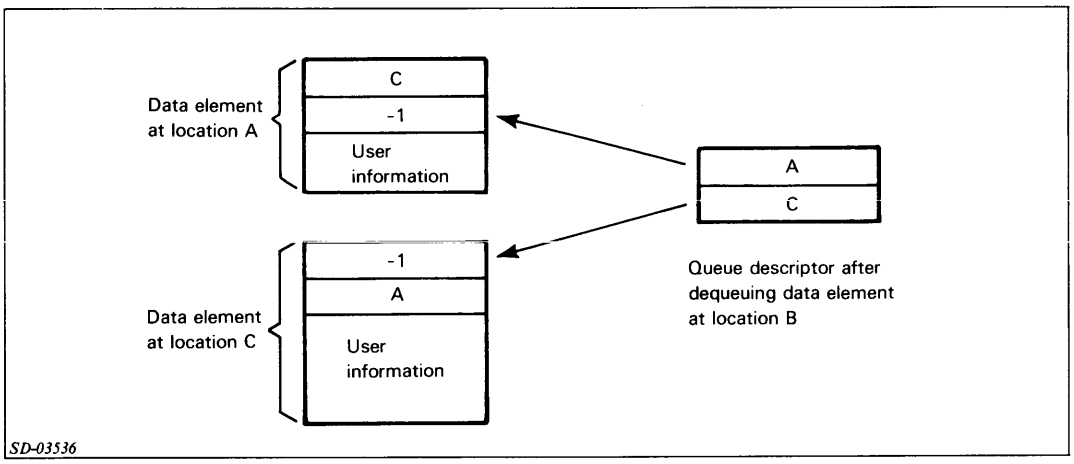


Figure 6.6 Data element dequeued

Queue Instructions

Table 6.3 lists the instructions for manipulating queues. Two of the instructions enqueue data elements onto queues, and one instruction dequeues data elements. The remaining instructions perform queue or queue-like searches.

Instruction	Operation
ENQH	Enqueue towards the head; add a data element to queue
ENQT	Enqueue towards the tail; add a data element to queue
DEQUE	Dequeue a queue data element; delete a data element
NBStc	Narrow search queue backward; 16-bit test condition
NFStc	Narrow search queue forward; 16-bit test condition
WBStc	Wide search queue backward; 32-bit test condition
WFStc	Wide search queue forward; 32-bit test condition
WMESS	Wide mask, skip and store if equal

Table 6.3 Queue instructions

Chapter 7
Device Management

Overview

The processor supports devices that transfer data using a slow, medium, or high speed transfer rate. With a programmed I/O facility, the processor transfers 1 or 2 bytes of data between a device and an accumulator. With a data channel I/O facility, the processor transfers blocks of words between a medium speed device and memory. With a burst multiplexor channel I/O facility, the processor transfers blocks of words between a high speed device and memory.

For instance, an asynchronous line controller transfers data with the programmed I/O facility. Medium speed devices, such as line printers and magnetic tapes, transfer data with the data channel I/O facility. Finally, the high speed disks transfer data with the burst multiplexor channel I/O facility.

Depending upon the operating system, you usually access a device through a system call to an operating system. The remainder of the Device Management chapter presents the basic information to assist you in reading and writing an interrupt handler or a device driver, which you invoke with a system call.

Device Access

The processor accesses a device through a programmed I/O facility, a data channel I/O facility, or a burst multiplexor channel I/O facility with the address translation disabled or enabled. For the processor to access a device with the address translation disabled, the processor ignores bits 2 and 3 of the segment base register. For the processor to access a device with the address translation enabled, bits 2 and 3 must first be set to enable the I/O instruction execution.

- Bit 2 is the LEF or I/O mode.

  Bit 2 specifies how the processor interprets the LEF and I/O instruction opcodes. For instance, in a segment where the processor executes LEF instructions, bit 2 must be set to one -- selecting the *LEF mode*. Thus, the processor interprets and executes the I/O and LEF instructions as LEF instructions.

  Conversely, in a segment where the processor executes I/O instructions, bit 2 must be set to zero -- selecting the *I/O mode*. Thus, the processor interprets I/O instructions and LEF instructions as I/O instructions. (Executing an I/O instruction requires an additional interpretation of bit 3.)

  NOTE: *Bit 2 affects the LEF instruction but not the ELEF, XLEF, and LLEF instructions.*

- Bit 3 is the I/O validity flag.

  Bit 3 enables or disables executing an I/O instruction. For instance, in a segment where the processor executes I/O instructions, bit 3 must be set to one. If bit 3 equals zero, the processor detects a protection violation when attempting to execute an I/O instruction.) Refer to the Memory and System Management chapter for servicing a protection fault.

You set up a data channel or burst multiplexor channel transfer with a program that specifies to a device driver (and to the device).

- The I/O channel to use for the transfer.

  In a multi-channel environment, the system uses the default I/O channel with 16-bit I/O instructions. The PRTSEL (NIO 3, CPU) instruction can be used to change the default I/O channel. In a single-channel environment, the PRTSEL instruction is a no-op.

  NOTE: *On power-up or after a system reset, channel 0 is the default channel. An I/O reset does not change the default.*

- The direction of the transfer (read or write).
- The address of the first word to transfer.

  The device transmits a word address to a device map. A *device map* is a set of map registers that control the addressing of memory for the data transfer.
- The total number of words to transfer.

The data channel or burst multiplexor channel facility uses a device map in either an unmapped or a mapped mode.

In the unmapped mode, the processor passes the word address directly to memory, as a physical address. You can use the load physical address (LPHY) instruction to translate a logical address to a physical address and store it in an accumulator. (The logical address must be pointing to a higher number segment.) You can then send the physical address to the device, using an I/O instruction.

In the mapped mode, the processor uses the device map and the word address to translate the most significant bits of the logical address to a physical page number. The processor then concatenates the physical page number to the 10 least significant bits of the logical address to form the physical address.

NOTE: *Refer to the specific functional characteristics manual for a description of the map register assignments and formats.*

Once you initialize the device, the transfer takes place in two phases.

1. First, the device driver initializes a device map with the starting word address of the block or subblock to transfer, with the number of words to transfer, and with the direction of the transfer.

2. Second, the data channel or burst multiplexor channel facility transfers the data between the device and memory.

For large transfers, you repeat the two phases until the processor transfers total number of words.

Table 7.1 lists the I/O instructions that affect a device map (a data channel map or burst multiplexor channel map).

Instruction	Operation
CIO	Issues a read or write command to a register of a device map
CIOI	Issues a read or write command to a register of a device map
IORST *	Sets the status register bits 0, 2-15 to 0 and turns off data channel and BMC mapping
WLMP	Loads a series of double words into a device map
LPHY	Translates a logical address and loads the physical address in an accumulator, for use in the unmapped mode

Table 7.1 I/O instructions for data channel/BMC maps

The * identifies an ECLIPSE C/350 compatible instruction

General I/O Instructions

You control the devices with I/O instructions. A general set of I/O instructions provide device independent operations. A special set of I/O instructions communicate with the I/O controller, load a device map, or service a vector interrupt.

The general I/O instructions receive data, send data, and initialize or test a device flag. Table 7.2 lists the general I/O instructions.

Instruction	Operation
DIA*[f]* *	Data in A (from A buffer of device)
DIB*[f]* *	Data in B (from B buffer of device)
DIC*[f]* *	Data in C (from C buffer of device)
DOA*[f]* *	Data out A (to A buffer of device)
DOB*[f]* *	Data out B (to B buffer of device)
DOC*[f]* *	Data out C (to C buffer of device)
IORST *	I/O reset
NIO*[f]* *	No I/O transfer (initialize a BUSY/DONE flag)
PIO	Issue a programmed I/O command to a device
SKP*t* *	I/O skip (test a BUSY/DONE flag and skip on condition)

Table 7.2 General I/O instructions

The *[f]* and *t* defines the optional device flag handling.

The * identifies ECLIPSE C/350 compatible instructions.

Figure 7.1 illustrates the format for a general I/O instruction.

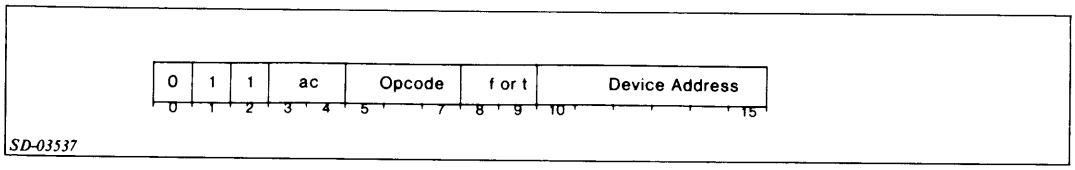


Figure 7.1 General I/O instruction format

where

011 — The 011 binary code indicates an I/O instruction.

ac — The ac field indicates a fixed-point accumulator (0-3).

The accumulator contains the data to send or to receive from a device.

Opcode — The opcode field identifies the I/O instruction operation.

Table 7.2 lists the I/O instructions.

f or t — The f bit identifies a device flag to change.

The t bit identifies a device flag to test.

Depending on the I/O instruction and the device, the instruction initializes or tests the device flag of the device. (See Tables 7.3 and 7.4.) For an external and an internal device (except for the CPU), the flags are *BUSY* and *DONE*. For the internal CPU device, the flags are interrupt on (*ION*) and power fail.

Device Address — The device address field identifies a unique device controller to send or to receive the data.

With a 6-bit device address, the processor can communicate with up to 64_{10} device controllers. The Assembler translates a standard three, four, or five letter device mnemonic to a device address.

Refer to the specific functional characteristics manual for a complete list of standard device mnemonics and for the corresponding device address.

Assembler Code for f	Bits 8 9	I/O Busy	I/O Done	CPU ION
(option omitted)	0 0	No effect	No effect	No effect
S	0 1	Set to a 1	Set to a 0	Set to a 1
C	1 0	Set to a 0	Set to a 0	Set to a 0
P	1 1	Pulses a special I/O bus control line		No effect

Table 7.3 Device flags for general devices

Assembler Code for t	Bits 8 9	I/O	CPU
BN	0 0	Test for BUSY = 1	Test for ION = 1
BZ	0 1	Test for BUSY = 0	Test for ION = 0
DN	1 0	Test for DONE = 1	Test for power fail = 1
DZ	1 1	Test for DONE = 0	Test for power fail = 0

Table 7.4 Device flags for skip instruction

The BUSY and DONE flags provide an indication of the device state to a device driver. When both flags equal zero, the device is idle. To start a device, you issue an I/O instruction with the proper device flag that sets the BUSY flag to one and the DONE flag to zero. When the device finishes the operation and becomes ready to start another operation, the device sets the BUSY flag to zero and the DONE flag to one.

The ION flag controls the device interrupt system. When the ION flag equals zero, the processor ignores interrupt requests. When the ION flag equals one, the processor services interrupt requests.

The power fail flag provides an indication of the processor state to the CPU device driver. When the power fail flag equals zero, the processor detects the proper power voltage ranges. When the power fail flag equals one, the processor detects a power fail.

Interrupts

The processor and an operating system maintain the I/O facilities through a hierarchical interrupt system. Any program can initiate an I/O operation by requesting a data transfer to or from a device. The program transmits the request through I/O system calls, which initialize the device and transfer the data by invoking the interrupt system.

The operating system maintains control of the interrupt system by manipulating an interrupt on flag, interrupt mask, and device flags. The interrupt on flag and interrupt mask reside in the processor. The interrupt on flag enables or disables all interrupt recognition, while the interrupt mask enables or disables selective device interrupt recognition.

The device flags reside in the device controller, and provide the interrupt communication link between the processor and the device. By manipulating the flags and the interrupt mask, the interrupt system can ignore all interrupt requests, or selectively service certain interrupt requests.

If the interrupt on flag and interrupt mask enable processor recognition of the interrupt request, the processor services the interrupt. To service an interrupt, the processor first determines the action to take on the currently executing instruction, then redefines the interrupt mask, and finally services the interrupt request. The Interrupt Servicing section explains the processor actions to transfer program control to the interrupt handler, and then to the interrupt service routine.

Interrupt On Flag

With the interrupt on (ION) flag equal to one, the processor responds to an interrupt request. When the ION flag equals zero, the processor cannot respond to an interrupt request.

You control the state of the ION flag with the INTDS and INTEN CPU device instructions. Refer to the specific functional characteristics manual for further information on the CPU device instructions.

Instruction Interruption

Most instructions are noninterruptible because they require only a minimum of CPU execution time. For instructions that require more time, such as the wide block move (WBLM) instruction, the processor (if required) interrupts the executing instruction, sets the processor status register bit 2 to one, and continues servicing the interrupt.

After servicing the interrupt, the processor either restarts or resumes the interrupted instruction. Refer to the specific instruction description and to the specific functional characteristics manual for further information on interruptible, restartable, and resumable instructions.

Interrupt Mask

A device is associated with one of the 16 bits in the interrupt mask. When the bit equals one, the mask blocks an interrupt request to the processor. When the bit equals zero, the processor services an interrupt request from the device. Since the processor can address greater than 16 device controllers, it can use a bit in the interrupt mask for two or more devices.

To change the state of a bit in the interrupt mask, use the mask out instruction (MSKO), which is a CPU device instruction.

Interrupt Servicing

To service an interrupt request (see Figure 7.2), the processor

1. Sets the ION flag to zero.
2. Determines if the address translation facilities are enabled.

   Refer to the System and Memory Management chapter for enabling and disabling the address translation facilities.
3. Fetch the pointer to the interrupt handler.

4. Changes the current segment of execution to segment 0, if the current segment equals segment 1 through 7 and if the address translation facilities are enabled.
5. Resolves the effective address of the interrupt handler.
6. Examines the first word of the interrupt handler, which is either
   - A 32-bit processor instruction (type 1).

     A 32-bit processor instruction contains bit 0 equal to one and bits 12-15 equal to 1001_2.
   - A 16-bit processor instruction (type 2).

     The processor identifies a 16-bit processor instruction as any instruction other than an XVCT or a type 1 instruction.
   - A vector interrupt (XVCT) instruction (type 3).
7. Stores the return address
   - Into logical locations two and three of segment 0 for the type 1 instruction.
   - Into location 0 of segment 0 for the type 2 instruction.
   - Into the vector stack (as part of the return block) for the XVCT instruction, during the vector interrupt processing.
8. Jumps indirectly
   - To the immediate interrupt handler and executes the type 1 instruction as the first instruction of the handler.

     The last instruction of the immediate interrupt handler must be a jump (XJMP or LJMP) instruction to jump indirectly through the return address. Refer to the specific functional characteristics manual for further information on reserved memory.
   - To the ECLIPSE C/350 interrupt handler and executes the type 2 instruction as the first instruction of the ECLIPSE C/350 interrupt handler.
   - To the vectored interrupt handler and executes the XVCT instruction.

     The last instruction of the vectored interrupt handler must be a wide restore from vector interrupt instruction (WRSTR), which pops the wide return block from the vector stack.

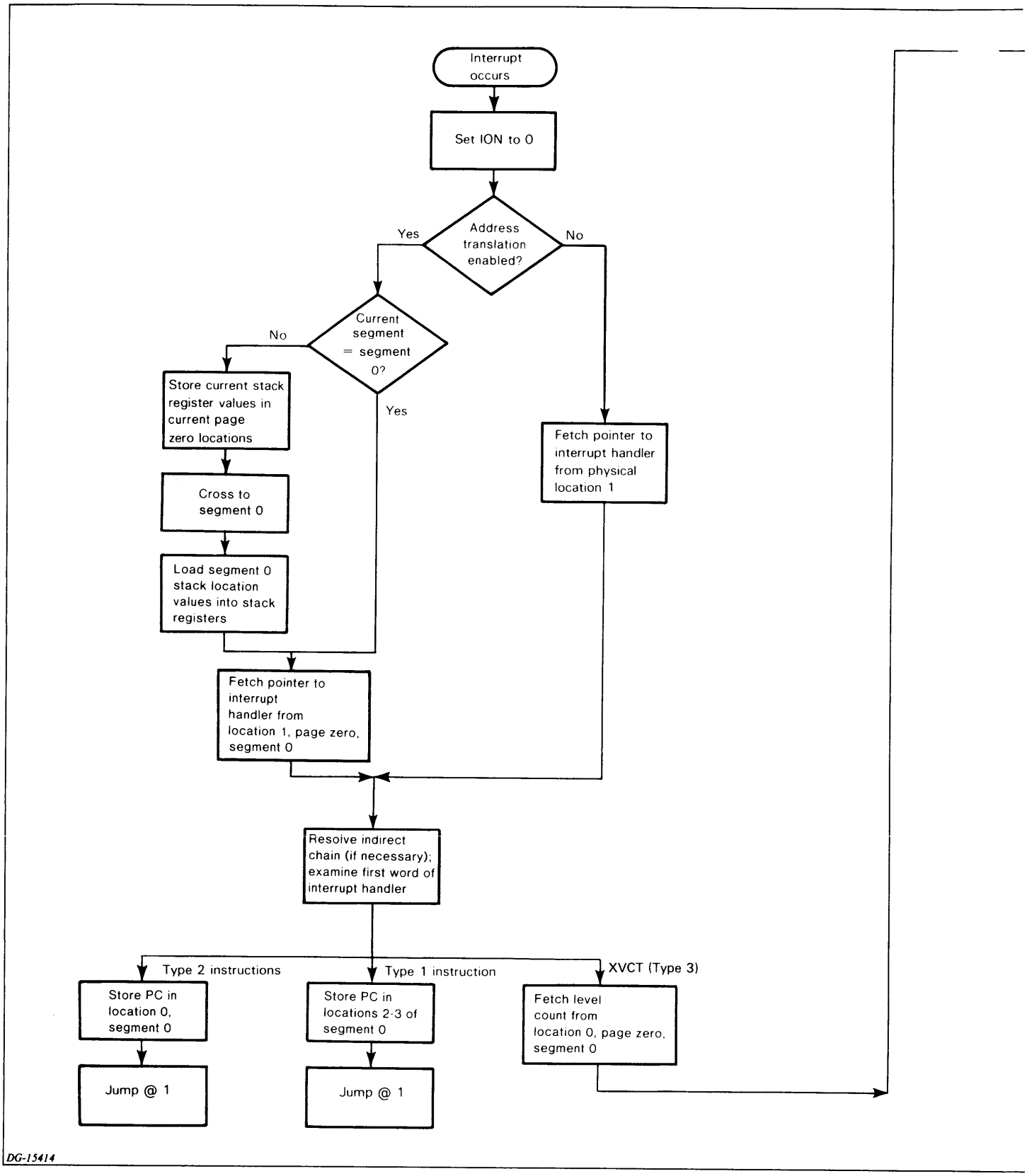
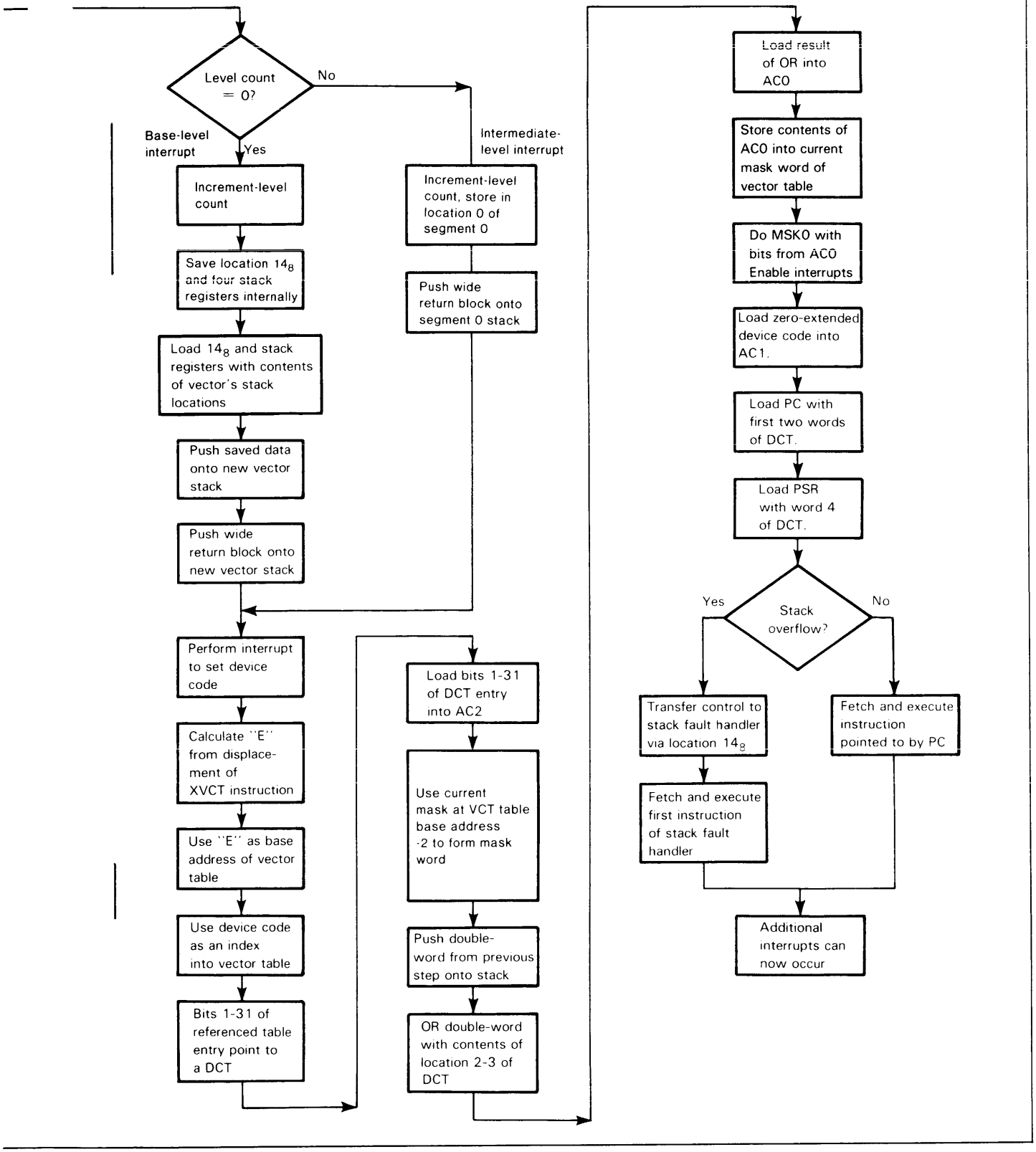


Figure 7.2 Interrupt sequence (continues)


Level count = 0?
No
Yes
Base-level interrupt
Intermediate-level interrupt
Increment-level count
Save location 14_8 and four stack registers internally
Load 14_8 and stack registers with contents of vector's stack locations
Push saved data onto new vector stack
Push wide return block onto new vector stack
Increment-level count, store in location 0 of segment 0
Push wide return block onto segment 0 stack
Perform interrupt to set device code
Calculate "E" from displacement of XVCT instruction
Use "E" as base address of vector table
Use device code as an index into vector table
Bits 1-31 of referenced table entry point to a DCT
Load bits 1-31 of DCT entry into AC2
Use current mask at VCT table base address -2 to form mask word
Push double-word from previous step onto stack
OR double-word with contents of location 2-3 of DCT
Load result of OR into AC0
Store contents of AC0 into current mask word of vector table
Do MSK0 with bits from AC0 Enable interrupts
Load zero-extended device code into AC1.
Load PC with first two words of DCT.
Load PSR with word 4 of DCT.
Stack overflow?
Yes
No
Transfer control to stack fault handler via location 14_8
Fetch and execute first instruction of stack fault handler
Fetch and execute instruction pointed to by PC
Additional interrupts can now occur

Vectored Interrupt Processing

The processor tests the contents of the interrupt-level word in reserved memory. If the contents equal zero, then the processor begins base-level interrupt processing. If the contents equal nonzero, then the processor begins intermediate-level interrupt processing. The processor, in either case, increments the contents by one.

NOTE: *Software, as part of the interrupt return, must decrement the interrupt-level word by one.*

Base-Level Interrupt Processing

The processor begins base-level interrupt processing at step 1 when the current segment equals segment 1 through 7. The processor begins base-level interrupt processing at step 5 when the current segment equals segment 0.

To service a base-level vector interrupt, the processor

1. Saves the wide stack pointer and the wide frame pointer in the reserved memory locations of the current segment. (The wide stack base and wide stack limit contents are the same as the reserved memory contents.)
2. Crosses to segment 0.
3. Saves the wide stack parameters from the reserved memory locations of segment 0 in an internal processor state.
4. Continues execution with step 6.
5. Saves the pointer to the wide stack fault handler and the four wide stack registers in an internal processor state.
6. Uses the three vector stack parameters in reserved memory to initialize the four wide stack registers and wide stack fault pointer.

    - Vector stack pointer parameter

      The processor, interpreting the parameter as a 16-bit word, zero extends the vector stack pointer before loading it into the wide stack base, wide stack pointer, and wide frame pointer registers.

    - Vector stack limit parameter

      The processor, interpreting the parameter as a 16-bit word, zero extends the vector stack limit before loading it into the wide stack limit register.

    NOTE: *The 16-bit vector stack base and limit parameters initially restrict the vector stack to the lower 128 Kbytes of segment 0.*

    - Vector stack fault address parameter

      Loading the vector stack information enables vector stack underflow and overflow detection.

7. Pushes the previously saved wide stack parameters from the internal processor state onto the vector stack.
8. Pushes a wide return block onto the vector stack.
9. Continues execution as the Final Interrupt Processing section explains.

Intermediate-Level Interrupt Processing

The processor begins intermediate-level interrupt processing with the current segment equal to segment 0. To service an intermediate-level vector interrupt, the processor

1. Pushes a wide return block onto the vector stack.
2. Continues execution as the Final Interrupt Processing section explains.

Final Interrupt Processing

To complete the vector interrupt servicing (see Figure 7.3), the processor

1. Calculates the effective address from the displacement of the XVCT instruction. The indirection chain, if any, is narrow.

   The effective address identifies word zero of the vector table. The table contains 64 double word entries for each I/O channel, one entry for each device on an I/O channel. Figure 7.4 illustrates the vector table.

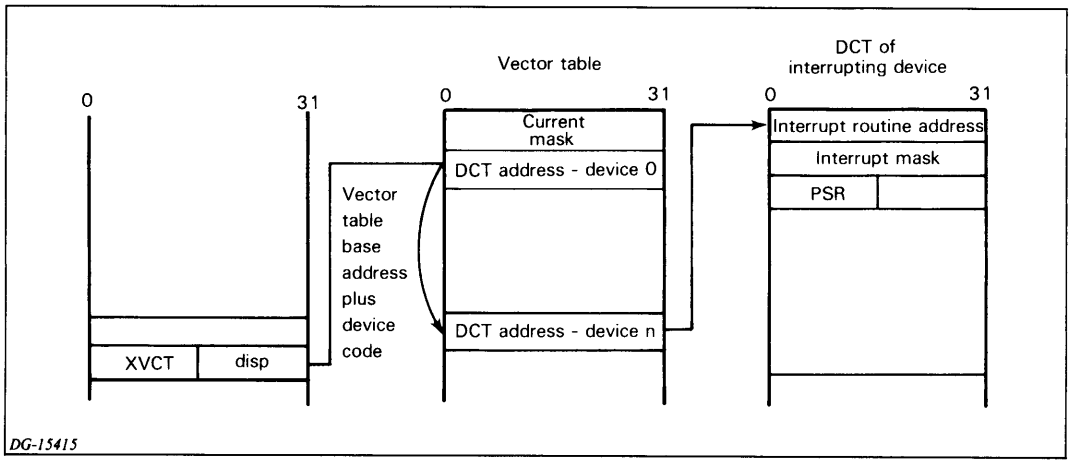


Figure 7.3 Sequence of actions to conclude interrupt service

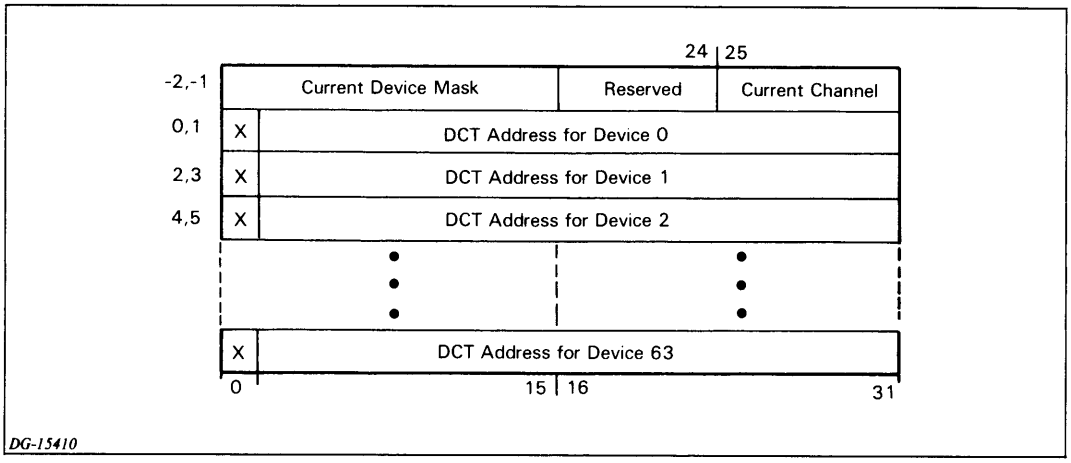


Figure 7.4 Vector table

2. Uses the interrupting device number as a double-word offset from the base of the vector table to address an entry.

   Bits 1-31 of the vectored entry contain the base address of a device control table (DCT). To satisfy the processor accesses to the device control table, you must construct the first five words as shown in Figure 7.5. In addition, you can build the device control table with more words to store device-dependent variables and constants for use by the device interrupt routine.

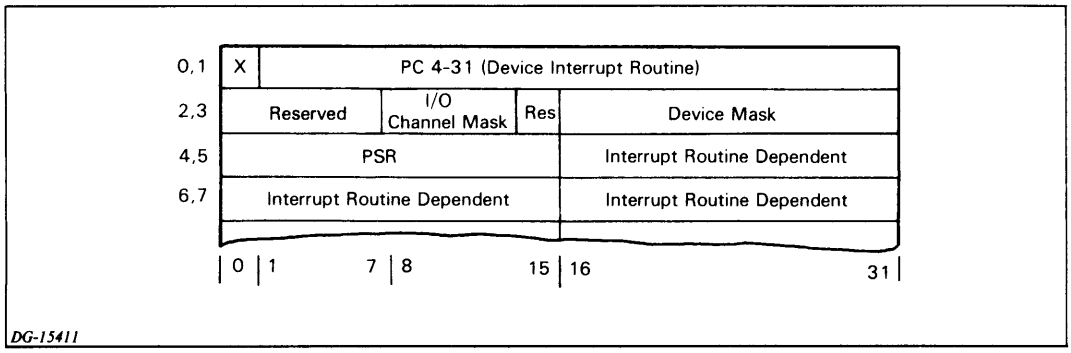


Figure 7.5 Device control table (DCT)

3. Loads AC2 with the base address of the device control table.

4. Constructs a double word and pushes it onto the vector stack.

   Bits 0-7 of the double word contain all zeros. Bits 8-31 contain the contents of the current mask from the vector table. The I/O channel masks are organized one bit per channel; for example, bit 14 equals I/O channel 7.

5. Loads AC0 with the inclusive or of the pushed double word and the second double word (words two and three) of the device control table.

6. Stores AC0 into the current mask.

7. Performs the function of a mask out (MSKO) instruction with AC0 and enables interrupts.

   When a mask bit equals one, the processor disables interrupt recognition of devices that use the mask bit.

8. Loads the least significant bits of AC1 with the interrupting I/O channel and device number, zero extended to 32 bits.
9. Loads the program counter with the address of the device interrupt routine (words zero and one of the device control table).
10. Initializes the PSR from the contents of the PSR word in the device control table.
11. Checks for a vector stack overflow.

    If the processor does not detect a vector stack overflow, it continues with step 12.

    If the processor detects a vector stack overflow, it transfers program control to the vector stack fault handler. The processor executes the first instruction of the vector stack fault handler before honoring further interrupts.
12. Executes the instruction addressed by the program counter.

    The processor executes the first instruction of the interrupt or vector stack fault handler before honoring further interrupts.

The processor requires that the pointer chain -- from the interrupt handler, to the vector table, to the device control table, and finally to the interrupt routine -- remain in Segment 0.

Chapter 8
Memory and System Management

Overview

The processor supports memory management and system management facilities for an operating system. The Memory and System Management chapter presents the basic information to assist the reading and writing operating system software.

The memory management facilities transform a logical address into a physical address and monitor the contents of the physical memory. The system management facilities return or modify implementation-dependent information about the system and the service faults.

The processor supports a virtual memory size of 4 Gbytes, which the processor distributes through eight segments. Each segment can support up to 512 Mbytes of logical address space. Since the logical address space is larger than the physical address space, the processor uses a demand-paging scheme.

The processor maintains pages of logical memory on disk until it needs them in the physical memory. (A page equals 2 Kbytes.) When referring to an instruction or to data that currently resides on disk, the processor moves the page to physical memory. However, when physical memory is full, the processor may first copy a page from memory to disk before moving the referenced page into memory. To facilitate the operation, the processor maintains tables in memory that determine

- Where a page resides (memory or disk resident).

  Bits 13-31 of a segment base register specify a physical address of a page table in memory. Each segment contains a page table, which occupies at least 2 Kbytes and begins on an integral 2 Kbyte boundary. A page table contains entries that indicate where the pages reside in memory.
- When to overwrite a page in memory with a page from disk.

  The processor maintains a table of referenced and modified bits.

The Memory and System Management chapter presents the memory management functions (segment access and address translation), and the system functions (processor identification and fault handling of privileged violations).

Segment Access and Address Translation

To access a memory word or words, the processor accesses a segment, translates a logical address (indirect or effective address) to a physical address, and accesses the physical page, which contains the word or words.

The following paragraphs describe the segment base registers, page tables, and the logical address to physical address translation.

Segment Base Registers

For the processor to access a segment, it first checks the segment base register specified in the logical address. Bit 0 of the segment base register controls access to the segment by specifying if the processor can refer to the segment for the instruction execution. If the processor cannot refer to the segment, the processor aborts executing the instruction and services a segment validity protection fault. Refer to the Protection Violations section in the Program Flow Management chapter for further information on protection fault handling.

The processor maintains eight segment base registers (SBR0 to SBR7) -- one for each of the eight segments. A segment base register contains information which

- Validates the segment access.
- Validates an I/O access.
- Specifies a one- or two-level page table.
- Specifies for the segment the address of the first entry in the page table.

You can modify the segment base registers with the LSBRA and LSBRS privileged instructions, which load a block of double words from memory into the segment base registers.

NOTE: *The processor must execute a privileged instruction in segment 0; otherwise, a protection violation occurs.*

Figure 8.1 shows the format of a segment base register.

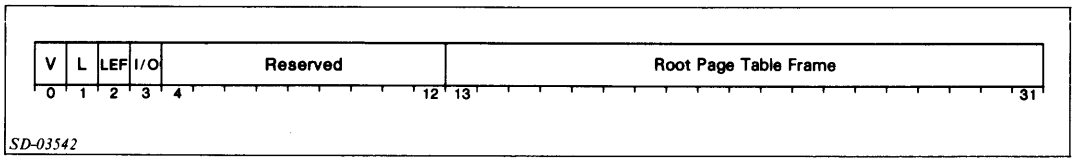


Figure 8.1 Segment base register format

where

V — The V bit is the segment validity flag.

The processor accesses the segment either to execute an instruction or to access data for an instruction that reads or writes data. However, the segment must be a valid reference.

The flag equals zero for an invalid segment.

The processor aborts executing a memory reference instruction and services a protection violation when the logical address refers to an invalid segment.

The flag equals one for a valid segment.

Following a valid segment check, the processor checks for a valid addressing range (translation level) in the logical address.

L — The L bit is a translation level flag.

The processor can access the segment with either a one-level or two-level page table.

The flag equals zero for a one-level page table.

The processor can use a one-level page table with a program that requires 1 Mbyte or less of logical address space in the segment. A one-level page table entry contains the page table offset for the physical address translation.

The flag equals one for a two-level page table.

The processor must use a two-level page table with a program that requires from 1 Mbyte to 512 Mbytes of logical address space in the segment. A two-level page table entry contains the address of the second page table, which contains the page table offset for the physical address translation.

Refer to the Page Table section for additional information on the page table. Refer to the Address Translation section for an example of using a segment base register and one or two page tables

LEF — The LEF bit is a LEF mode flag.

The processor checks the LEF flag when executing a load effective address instruction (LEF) and an I/O instruction.

The flag equals one for a LEF instruction interpretation.

The processor executes the instruction as a LEF instruction.

The flag equals zero for an I/O instruction interpretation.

Before executing the instruction as an I/O instruction, the processor checks the I/O validity flag.

I/O — The I/O bit is an I/O validity flag.

The processor checks the I/O validity flag when executing an I/O instruction.

The flag equals zero for an illegal I/O operation.

The processor aborts executing the I/O instruction and services the protection violation.

The flag equals one for a legal I/O operation.

The processor executes the I/O instruction.

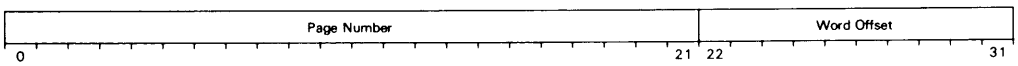
Reserved — DGC reserves bits (4-12) for internal DGC use. Refer to the specific functional characteristics manual for additional information.

Root Page Table Frame — The root page table frame (frame number) (bits 13-31) specifies the 19 most significant bits of the physical address for the root page table page. (The table begins on a 2 Kbyte address boundary.) The remaining bits of the address come from either bits 4-12 or 13-21 of the logical address.

Pageframes

A pageframe is a page address shifted right ten bits.

A physical address has the following format:



A page address is considered to be the page number with ten zeroes following it. This page number is also the pageframe, because the page actually includes the addresses between the start of the page (word offset 0) and the end of the page (word offset $3FF_{16}$. For example:

Page 1 is: 1 0000000000 = 400_{16}
Page 55 is: 0101 0101 0000000000 = 15400_{16}

Pageframe 1 corresponds to the page address 400_{16} and refers to the data words with addresses 400_{16} through $7FF_{16}$.

Page Tables

In each segment, the processor accesses a page table that specifies the status of the pages for the segment in memory. The page table manipulation instructions are load page table entry (LPTE) and store page table entry (SPTE). The page table contains an entry (PTE) for each page, which

- Indicates if a page is a valid access and the type of access.
- Indicates if a page is currently in physical memory.
- Contains information needed to translate a logical address to a physical address.

Figure 8.2 shows the format of a page table entry.

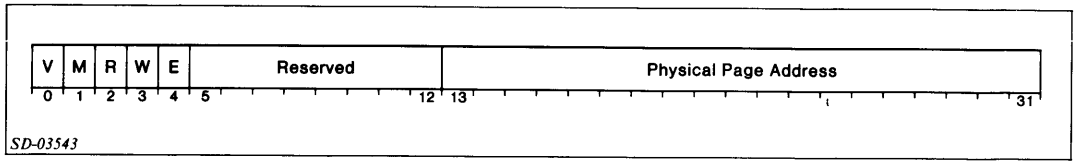


Figure 8.2 Page table entry format

where

V — The V bit is the valid access flag.

The processor accesses the page to read or write data, or to execute an instruction. However, the page must be a valid reference.

The flag equals zero for an invalid page.

The processor aborts executing the memory reference instruction and services the protection violation when the logical address refers to an invalid page.

The flag equals one for a valid page.

Following the valid page check, the processor checks for a valid page access (read, write, or execute).

M — The M bit is the memory resident flag.

For the processor to access a page for reading or writing data, or for executing an instruction, the page must reside in memory.

The flag equals zero for a disk-resident page.

The processor suspends executing the memory reference instruction and services the page fault when the logical address refers to a disk-resident page. Following the page fault, the processor resumes executing the memory reference instruction.

The flag equals one for a memory-resident page.

The processor completes executing the memory reference instruction when the logical address refers to a memory-resident page.

R — The R bit is the read access flag.

The processor accesses the page to read data from memory.

The flag equals zero for a page that the processor cannot gain read access.

The processor aborts executing the memory reference instruction and services the protection violation when the instruction requests a read operation, such as loading an accumulator or skipping on the condition of a memory word.

The flag equals one for a page that the processor can gain read access.

Following the valid read access, the processor checks for a disk- or memory-resident page status.

NOTE: *A page with write or execute access also requires read access; otherwise, results are indeterminate.*

W — The W bit is the write access flag.

The processor accesses the page to write data into memory.

The flag equals zero for a page that the processor cannot gain write access.

The processor aborts executing the memory reference instruction and services the protection violation when the instruction requests a write operation, such as storing an accumulator or modifying a bit of a memory word.

The flag equals one for a page that the processor can gain write access.

Following the valid write access, the processor checks for a disk- or memory-resident page status.

E — The E bit is the execute access flag.

The processor accesses the page to execute an instruction.

The flag equals zero for a page that the processor cannot gain execute access.

The processor aborts executing the memory reference instruction and services the protection violation when the processor cannot execute the instruction.

The flag equals one for a page that the processor can gain execute access.

Following the valid execute access, the processor checks for a disk- or memory-resident page status.

NOTE: *The processor ignores the page access bits (bits 2-4) for a page table entry that addresses another page table page, which occurs during a two-level page table translation.*

Reserved	DGC reserves bits 5-12 for internal DGC use. Refer to the specific functional characteristics manual for additional information.
Physical Page Address	The physical page address (bits 13-31) identifies a page in memory. The physical page address refers to a page containing an instruction and/or data, or refers to a page containing the base of another page table, as determined by a one- or two-level page table translation.

Address Translation

Following a valid segment reference, the processor checks the range of the logical address space within the segment, and compares it to the address range of the logical address. Bit one of the segment base register defines a one- or two-level page table, which specifies the addressing range. (Refer to the Segment Base Register section for further details.)

The processor compares bit 1 of the segment base register with bits 4-12 of the logical address. When bit 1 equals a zero, the logical address bits 4-12 must be all zeros. The processor aborts executing the instruction and services the protection fault (page table depth fault) when any of the logical address bits 4-12 contain a one.

NOTE: *A page table depth fault can occur when a program that was allocated 1 Mbyte of memory attempts to access a location beyond the 1 Mbyte boundary.*

Figure 8.3 illustrates an indirect or an effective logical address for a one- and two-level page table. Refer to the System Overview chapter for an explanation of calculating an indirect or effective logical address.

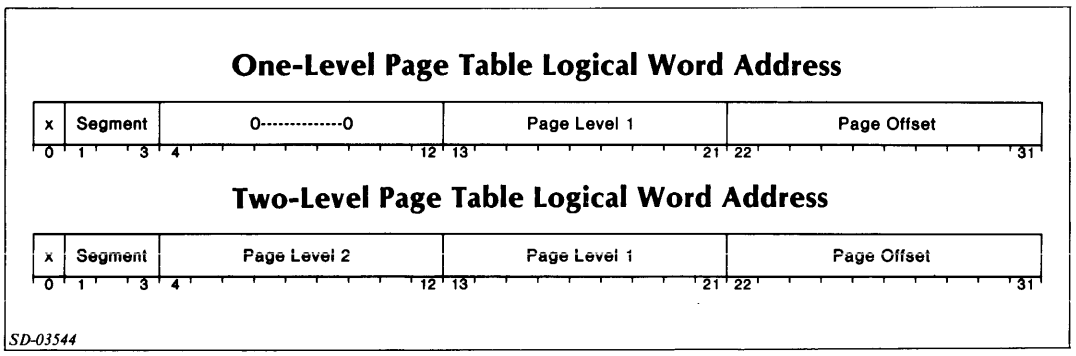


Figure 8.3 Indirect and effective logical address formats

where

x	The x bit (bit 0) is ignored by the processor when using direct addressing. The processor tests the x bit when using indirect addressing, and continues testing the bit in subsequent indirect address until the bit equals zero.
Segment	The segment (bits 1-3) specifies one of eight segment base registers.

Page Level 2

The page level 2 (bits 4-12) specifies an entry in the first of two page tables for a two-level page table translation. The page table entry contains the address of the second page table.

For a one-level page table translation, the page level 2 field (bits 4-12) must be all zeros. If the bits are not zero, the processor aborts the instruction and services a page table validity protection fault. Refer to the Protection Violations section in the Program Flow Management chapter for further information on protection fault handling.

Page Level 1

The page level 1 (bits 13-21) specifies an entry in a page table

For a one- or two-level page table translation, the page table entry contains the address of the final page (to be accessed for data or an instruction).

Page offset

The page offset (bits 22-31) specifies the final entry in the final page.

The page offset completes the address translation.

The Address Translation section presents examples of a one- and a two-level page table translation. The circled numbers labeling the accompanying paragraphs correspond to the circled numbers shown in Figures 8.4 and 8.5. Figure 8.4 illustrates a one-level page table translation. Figure 8.5 illustrates a two-level page table translation.

Page Access

When an instruction refers to a page, the processor determines the validity of the access by checking the access request with the appropriate validation and access validation bits in the page table entry.

When an instruction refers to a valid page that is not currently in physical memory, a page fault occurs. The fault handler saves the current state of the processor in reserved memory (context block), moves a memory page to disk (if required), and then transfers the referenced page from disk to memory.

Access Validation

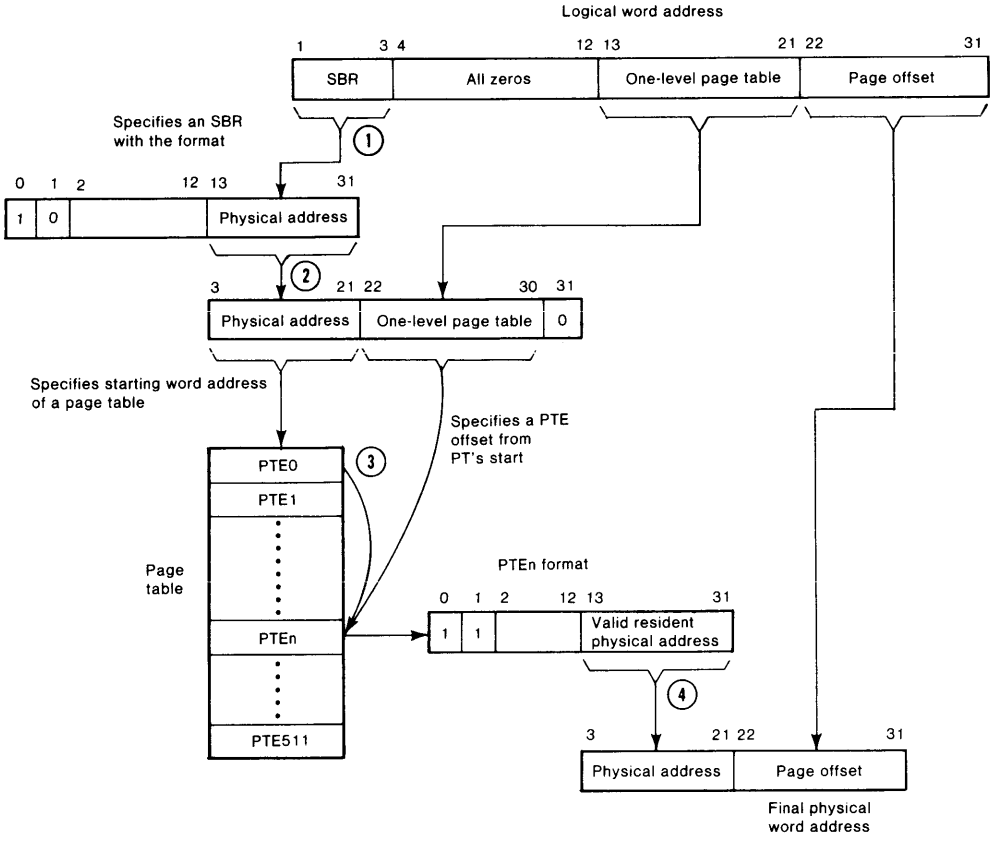
When a referenced page is valid, the processor determines whether the page is restricted to a particular access. Bits 2-4 of the referenced page table entry contain the access bits that specify any restriction.

When the reference to memory is for reading, the processor checks bit 2. A one in bit 2 indicates a valid read, while a zero indicates an invalid read. When the reference is invalid, a protection fault occurs and AC1 contains the error code 0.

NOTE: *In general, read access must* always *be available to any page with execute or write access.*

When the reference to memory is for writing, the processor checks bit 3. A one in bit 3 indicates a valid write, while a zero indicates an invalid write. When the reference is invalid, a protection fault occurs and AC1 contains the error code 1.

When the reference to memory is for executing, the processor checks bit 4. A one in bit 4 indicates a valid execute, while a zero indicates an invalid execute. When the reference is invalid, a protection fault occurs and AC1 contains the error code 2.



1. The logical word address to be translated has the format shown in the diagram. Bits 1-3 of the word address specify one of the eight segment base registers (SBRs). The processor uses the contents of this valid SBR to form the physical address of a PTE.

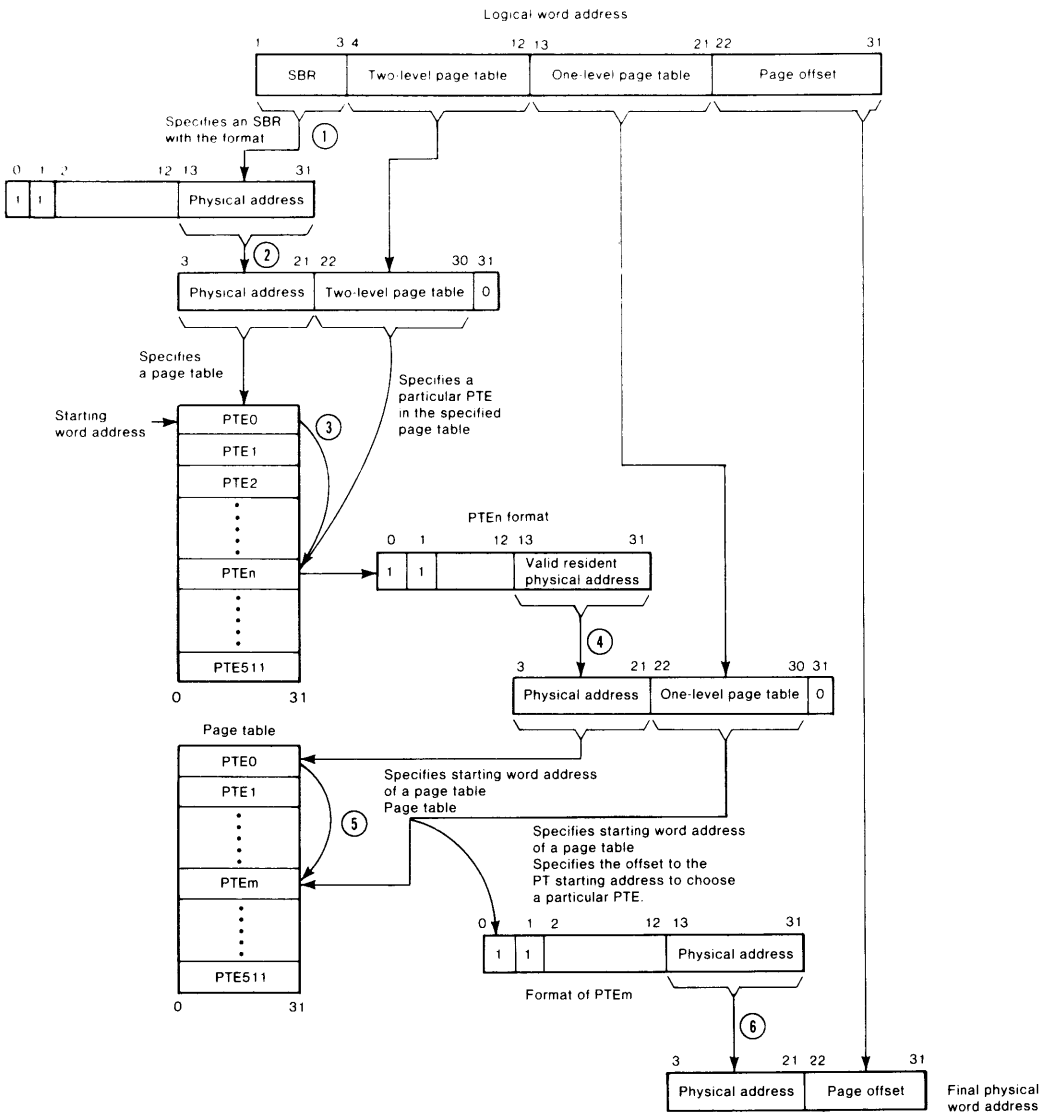
2. To form this physical page address, the processor begins with the physical address specified in bits 13-31 of the SBR. This address becomes bits 3-21 of the PTE address. Bits 13-21 of the logical word address become bits 22-30 of the PTE address. The processor appends a zero to the right of the PTE address, making a 29-bit word address.

3. Bits 3-21 of the PTE address (unchanged in 2 above) specify the starting address of a page table. Bits 22-31 of the PTE address specify an offset from the start of the table to some PTE (labeled PTE*n* in the figure). This PTE specifies the starting address of a page of memory.

4. PTE*n* bits 13-31, the page address, become bits 3-21 of the physical address. The page offset field specified in bits 22-31 of the logical word address become bits 22-31 of the physical address. This is the physical word address translated from the original word address.

SD-03545

Figure 8.4 One-level page table translation



(1) Logical word address to be translated has the format shown in the diagram. Bits 1-3 of the word address specify one of the eight segment base registers (SBRs). The processor uses the contents of this valid SBR to form the address of a PTE.

(2) To form this address, the processor begins with the physical address specified in bits 13-31 of the SBR. This address becomes bits 3-21 of the PTE address. Bits 4-12 of the logical word address become bits 22-30 of the PTE address. The processor appends a zero to the right of the PTE address, making a 29-bit word address.

(3) Bits 3-21 of the PTE address specify the starting address of a page table. Bits 22-31 of the PTE address specify an offset from the start of the table to some PTE (labeled PTE*n* in the figure). The PTE specifies the starting address of a page table.

(4) The processor now constructs the address of a second PTE. The physical address specified in bits 13-31 of the first (PTE*n*) become bits13-21 of the address of the second PTE*m*. Bits 13-21 of the logical word address become bits 22-30 of the second PTE's address. The processor appends a zero to the right of the second PTE address to make a 29-bit word address.

(5) Bits 3-21 of the second PTE address specify the starting address of a second page table. Bits 22-31 of the second PTE address specify an offset from the start of the second table to some PTE (labeled PTE*m* in the figure). The second PTE specifies the starting address of a page. The page table containing PTE*m* can be paged itself. PTE*n* can indicate a nonresident page table.

(6) The second PTE*m*'s bits 13-31, the page address, become bits 3-21 of the physical address. The page offset specified in bits 22-31 of the logical word address becomes bits 22-31 of the physical address. This last value is the physical word address.

SD-03546

Figure 8.5 Two-level page table translation

Demand Paging

Since the logical address space is larger than the physical memory space, all pages cannot reside in physical memory at the same time. A paging facility (under control of the page fault handler) moves referenced pages in and out of memory whenever necessary -- *demand paging.*

When an instruction refers to a valid page not currently in physical memory, then a page fault occurs. A status field in the context block indicates the cause of the page fault. Refer to the specific functional characteristics manual for more information on the context block. If an instruction refers to a location that requires a two-level page table when only a one-level page table is allocated, then a protection violation occurs. Refer to the Protection Violations section in the Program Flow Management chapter for more information.

To service the page fault, the processor

1. Saves the current state of the processor in reserved memory of segment 0 (context block).
2. Crosses to segment 0 if the current segment is 1-7.
    a. Stores the wide stack pointer and wide frame pointer contents into the page zero locations of the current segment.

       The values of the stack limit and stack base registers should be identical to the values in reserved memory.

    b. Redefines the wide stack for segment 0.

       The processor initializes the wide stack pointer, wide stack limit, and wide stack base registers from reserved memory of segment 0.
3. Executes a jump indirect through the pointer in reserved memory of segment 0 to the page fault handler.
4. Executes the page fault handler, which
    a. Initiates restoring a page from memory to disk (if necessary).

       Refer to the Referenced and Modified Flags section for more information on determining when a page needs to be restored to disk.

       The page fault handler invokes the interrupt system to transfer the page to disk.

    b. Initiates loading the referenced page from disk to memory after the page fault handler restores the referenced page to disk.

       The page fault handler invokes the interrupt system to transfer the page from disk.

    c. Restores the state of the processor after the page fault handler loads the referenced page into memory.

       The page fault handler executes the WDPOP instruction, which restores the state of the processor and restarts the interrupted program. The WDPOP instruction accesses the information in the context block to restore the processor state.

5. Completes the memory reference and continues executing the instruction.

NOTE: *A page fault must not occur during steps 1, 2, and 3; otherwise, the processor halts.*

Referenced and Modified Flags

A referenced flag and a modified flag are associated with a physical page in memory. When the processor reads a word from memory, it sets the referenced flag associated with the physical page to one. When the processor writes a word to memory, the processor sets the referenced and modified flags associated with the physical page to one. A read or write operation occurs when the processor accesses memory without a protection fault occurring on a memory resident page.

NOTE: *An I/O memory reference does not affect the state of the flags.*

The referenced flag helps to determine which page in physical memory the page fault handler should replace with a new page from disk. The referenced flag allows an operating system and the page fault handler to determine the frequency of references to individual pages.

The modified flag indicates if the processor wrote to a memory page. When a modified flag equals one, the processor modified the contents of the page. The page fault handler must first copy the page to disk before moving a new page from disk to memory. If a modified flag equals zero, the processor did not modify the contents of the page, and the page fault handler can immediately move a new page from disk to memory.

Table 8.1 lists the privileged instructions that manipulate the referenced and modified flags (or bits). Refer to the Fixed-Point Computing chapter for a list of additional instructions that manipulate bit strings.

Instruction	Operation
LMRF	Load modified and referenced bits
ORFB	OR referenced bits
PATU	Purge address translator
RRFB	Reset referenced bits
SMRF	Store modified and referenced bits

Table 8.1 Instructions that manipulate referenced and modified flags

Central Processor Identification

The processor stores information about the processor parameters (such as the memory size and the microcode revision level) in one or more fixed-point accumulators. Table 8.2 lists the central processor identification instructions. Refer to the specific functional characteristics manual for further information on the accumulators.

Instruction	Operation
ECLID	Load CPU identification (AC0, bits 0-31)
LCPID	Load CPU identification (AC0, bits 0-31)
NCLID	Narrow load CPU identification (AC0-AC2, bits 16-31)

Table 8.2 System identification instructions

Program Flow Instructions

Table 9.4 lists the C/350 program flow instructions that refer to memory. The table also lists an equivalent 32-bit processor instruction that you can substitute to expand (within the segment) the memory address range and to use the wide stack.

Unless otherwise stated, the C/350 instruction and the 32-bit processor equivalent instruction use identical

- Single or double word instruction length
- Argument string
- Data access for writing and for reading (register or memory)

The data precision changes from 16 to 32 bits.

However, an equivalent 32-bit processor instruction uses a double word indirect pointer, while the C/350 instruction uses a single word indirect pointer.

C/350 Instruction	Operation	Equivalent Instruction
DSPA	Dispatch	**LDSP**
EJMP	Extended jump	**XJMP**
EJSR	Extended jump to subroutine	**XJSR**
ELEF	Extended load effective address	**XLEF**
JMP	Jump	-
JMP ,1	Jump, relative to the program counter	**WBR**
JSR	Jump to subroutine	-
LEF	Load effective address	-
POPB	Pop block and execute (return from XOP0)	**WPOPB**
POPJ	Pop PC and jump (return with PSHJ)	**WPOPJ**
PSHJ	Push jump (return with POPJ)	**XPSHJ**
PSHR	Push return address (pop with POPJ)	-
RSTR	Restore (return from VCT -- mode E)	**WRSTR ****
RTN	Return	**WRTN ***
SAVE	Save (used with JSR)	**WSSVR*** **WSSVS ***
SAVZ	Save without arguments (used with JSR)	**WSSVR*** **WSSVS ***
XOP0 ***	Extended operation (return with POPB)	**WXOP *****

Table 9.4 C/350 program flow management instructions

*The WRTN, WSSVS, and WSSVR instructions modify the OVK fixed-point overflow mask and use a return block of six double words.

**The XVCT and WRSTR instructions use the wide stack, and are equivalent to RSTR and mode E of the VCT instruction.

***The XOP0 and WXOP instructions are double word instructions.

Stack Instructions

Table 9.5 lists the C/350 stack instructions that refer to memory. The table also lists an equivalent 32-bit processor instruction that you can substitute to expand (within the segment) the memory address range.

Unless otherwise stated, the C/350 instruction and the 32-bit processor equivalent instruction use identical

- Single or double word instruction length
- Argument string
- Data access for writing and for reading (register or memory)

The data precision changes from 16 to 32 bits.

However, an equivalent 32-bit processor instruction uses a double word indirect pointer, while the C/350 instruction uses a single word indirect pointer.

C/350 Instruction	Operation	Equivalent Instruction
MSP	Modify stack pointer	**WMSP**
POP	Pop multiple accumulators	**WPOP**
POPB	Pop block and execute (return from XOP0)	**WPOPB**
POPJ	Pop PC and jump	**WPOPJ**
PSH	Push multiple accumulators	**WPSH**
PSHJ	Push jump	**XPSHJ**
PSHR	Push return address	-
RSTR	Restore (return from VCT -- mode E)	**WRSTR ****
RTN	Return	**WRTN ***
SAVE	Save (used with JSR)	**WSSVR*** **WSSVS ***
SAVZ	Save without arguments (used with JSR)	**WSSVR*** **WSSVS ***
XOP0 ***	Extended operation (return with POPB)	**WXOP *****

Table 9.5 C/350 stack management instructions

*The WRTN, WSSVS, or WSSVR instructions modify the OVK fixed-point overflow mask and use a return block of six double words.

**The XVCT and WRSTR instructions use the wide stack, and are equivalent to RSTR and mode E of the VCT instruction.

***The XOP0 and WXOP instructions are double word instructions.

Chapter 9
C/350 Programming

Overview

The 32-bit processor executes 16-bit processor instructions to provide upward program compatibility and to develop 16-bit programs (for instance, for the ECLIPSE C/350 processor). The C/350 Programming chapter presents both issues.

Programs that include C/350 memory-referenced and C/350 stack-referenced instructions must meet certain requirements or restrictions explained in this chapter. The specific functional characteristics manual presents any additional machine restrictions. Refer to the *ECLIPSE C/350 Principles of Operation* manual for an explanation of C/350 instructions, terms, and conventions.

C/350 Registers

The C/350 fixed-point accumulator bits 0-15 correspond to the wide fixed-point accumulator bits 16-31. When a C/350 instruction loads data into an accumulator, it alters bits 16-31, and leaves bits 0-15 undefined, unless otherwise noted. When a C/350 instruction reads data from an accumulator (bits 16-31), it does not alter the contents (such as a CLM instruction).

The C/350 fixed-point accumulator bits 1-15 correspond to the wide accumulator bits 17-31 for accumulator relative addressing.

The C/350 instructions do not affect the processor status register.

The C/350 floating-point accumulators are identical to the 32-bit processor floating-point accumulators.

The C/350 program counter bits 1-15 correspond to the wide program counter bits 17-31. A C/350 program flow instruction modifies bits 17-31, while the most significant bits are the current segment and zeroes (see Figure 9.1).

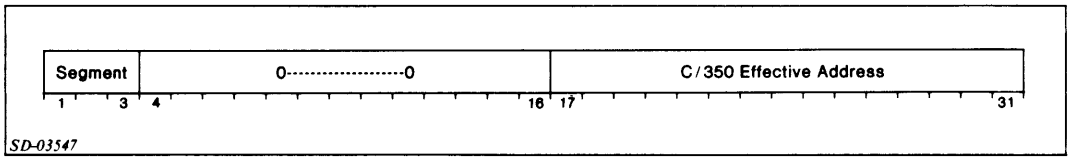


Figure 9.1 C/350 program counter format

where

Segment	The segment number specifies the current segment.
C/350 Effective Address	The C/350 effective address remains within the first 64 Kbytes of the segment.

C/350 Stack

The C/350 stack (or narrow stack) supports C/350 program development and upward program compatibility. Unlike the wide stack, the narrow stack uses three parameters (in reserved memory) to define and to control the narrow stack.

1. The narrow stack limit -- defines the upper limit of the narrow stack.

   Although specifying one word, the narrow stack limit functions like the wide stack limit.

2. The narrow stack pointer -- initially, defines the lower limit of the narrow stack.

   If you wish to enable narrow stack underflow, initialize the narrow stack pointer to 400_8 and start the narrow stack area at location 401_8.

   After accessing the narrow stack, the narrow stack pointer defines the current location of the last word written onto or read from the narrow stack. (Although specifying one word, the narrow stack pointer functions like the wide stack pointer.)

3. The narrow frame pointer -- defines a reference point in the narrow stack.

   Although specifying one word, the narrow stack frame pointer functions like the wide stack frame pointer.

The C/350 (or narrow) return block normally consists of five words: the contents of the least significant 16 bits of the four accumulators, the least significant 15 bits of the program counter or the frame pointer, and the carry in bit 0 of the last word pushed. The Program Flow Management chapter presents the narrow stack fault-handling. Refer to the *ECLIPSE C/350 Principles of Operation* manual for additional information.

C/350 Faults and Interrupts

The 32-bit processor services (with the same pointers and fault handlers) the 16- and 32-bit floating-point and decimal/ASCII faults. For floating-point faults, the processor pushes a return block onto either the narrow or the wide stack, depending on the first instruction of the floating-point fault handler (a 16- or 32-bit instruction). For decimal/ASCII faults, the processor pushes a return block onto either the narrow or the wide stack, depending on bit 0 of the fault code in AC1 (bit 0 equals one for C/350 faults). Thus, you can upgrade a program, written for the 16-bit processor to incorporate 32-bit processor enhancements. Refer to the Program Flow Management chapter for more information on the fault handlers.

The 32-bit processor services (with the same pointer and interrupt handler) the 16- and 32-bit I/O interrupts. The processor pushes a return block onto either the narrow or the wide stack, depending on the first instruction of the I/O interrupt handler (a 16- or 32-bit instruction). Thus, you can upgrade a program written for the 16-bit processor to incorporate 32-bit processor enhancements. Refer to the Device Management chapter for more information on the interrupt handler.

The 32-bit processor and the 16-bit processor use different methods to flag an interrupted and resumable EDIT instruction. While the 16-bit processor sets AC0 to minus one (177777_8), The 32-bit processor sets a resume flag (IRES) in the PSR and checks the flag after completing the interrupt. For compatibility purposes, the 32-bit processor also sets AC0 to minus one.

Expanding an ECLIPSE C/350 Program

You can expand a 16-bit program by using a specific set of 32-bit instructions to

- Expand the program beyond 64 Kbytes.
- Use expanded data areas, such as large arrays.
- Utilize the 32-bit fixed arithmetic.

There are several methods available to expand a 16-bit program beyond 64 Kbytes. The most reliable approach is to rewrite one of the subroutines to contain 32-bit instructions, and place it in the segment anywhere above the lower 64 Kbytes. The program must call the expanded subroutine with the XJSR or LJSR instruction. The subroutine must begin with a wide special save (WSSVR or WSSVS) instruction and end with a wide return (WRTN) instruction. Also, the subroutine must use the 32-bit memory-reference instructions.

To expand data areas for large arrays or buffers, the processor must perform address calculations with the 32-bit fixed integer arithmetic, and it must reference data with the 32-bit memory-referenced instructions. You must then change the program to refer to the expanded data area.

You can also create additional subroutines to maintain the large arrays and to reference the data through these routines. If you write an additional subroutine, ensure that you refer to the subroutine with the wide special save and wide return instructions. (The use of SAVE and RTN result in the loss of bits 0 to 15 of the accumulators and the processor status register.)

To use the 32-bit fixed point arithmetic, all operations on the data (loading, calculations, and storing) must be performed with the 32-bit instructions. This can be accomplished with spot changes or through new subroutines, but again, care must be taken when mixing these operations with 16-bit operations.

Expanding an ECLIPSE C/350 Subroutine

You can call a C/350 subroutine from a 32-bit routine with the changes listed in Table 9.1.

Changes to C/350 Subroutine	Reason for Change
Replace SAVE and RTN with WSSVS or WSSVR and WRTN	A routine can call the subroutine from an address, which exceeds 16 bits. Also, the accumulators can contain 32-bit entities.
Check external references for 32-bit memory reference instructions	A routine could pass 32-bit fixed-point data. Also, a called lower level subroutine can be located in an address space, which exceeds 16 bits.
Check short negative references on the stack that may require 32-bit displacements	Using WSSVS or WSSVR in this subroutine changes the size of the pushed stack block, requiring the Assembler to recalculate the negative reference.
Change a routine (to save the 31-bit PC) that calls a subroutine with a JSR through page zero	A long address requires 31 bits, and can cause the program to run out of page zero locations. Use LJSR or XJSR in the calling routine to save the 31-bit PC.

Table 9.1 Alternations to C/350 subroutines

C/350 Instructions

The C/350 Instructions section presents the instructions that refer to memory or to the narrow stack. The remaining C/350 instructions (such as ADD) are presented with the other 32-bit processor instructions. The ECLIPSE *MV/Family Instruction Reference Booklet* identifies the C/350 instructions supported on the 32-bit processors. Refer to the specific functional characteristics manual for any additional instructions.

C/350 Memory Reference Instructions

The processor considers the C/350 memory reference instructions to be within the first 32 Kwords (64 Kbytes) of the current segment. If the processor executes a C/350 memory reference instruction above the 32 Kword limit, the effective address reverts to within the C/350 address space (lower 32 Kwords).

To refer to a word with a C/350 memory reference instruction, the processor forms an effective address (see Figure 9.2).

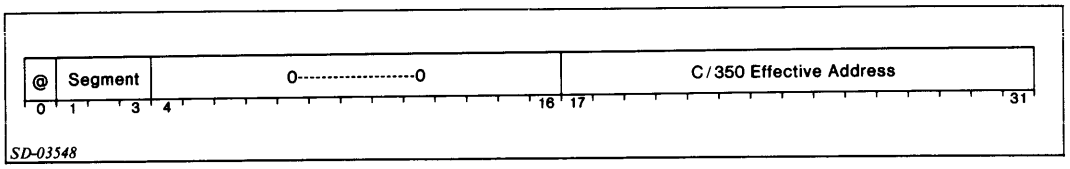


Figure 9.2 C/350 word addressing format

where

@ The indirect (@) bit in bit 0 forces indirect addressing (when set to one), through a single word pointer.

Segment The segment number specifies the current segment.

Figure 9.3 illustrates the C/350 effective addressing.

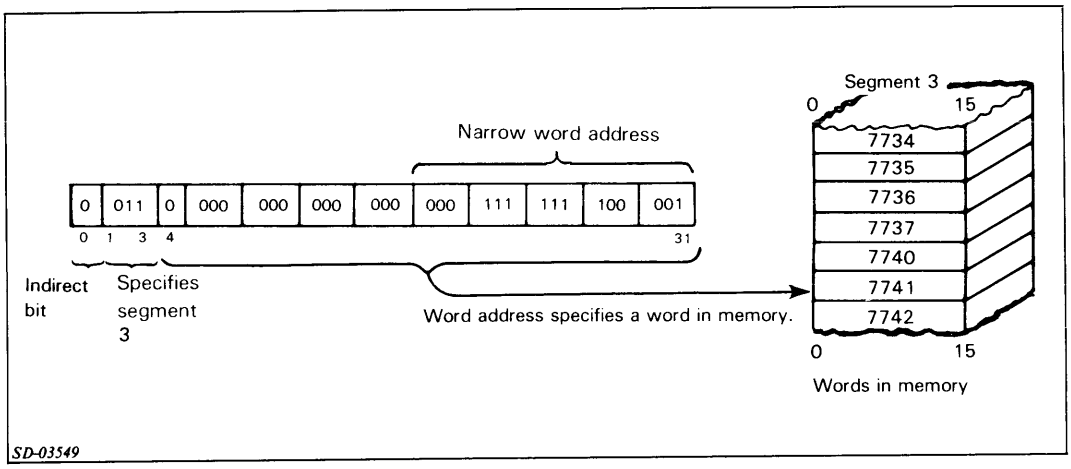


Figure 9.3 C/350 effective addressing

To refer to a byte, with a C/350 memory reference instruction, the processor forms a byte address (see Figure 9.4).

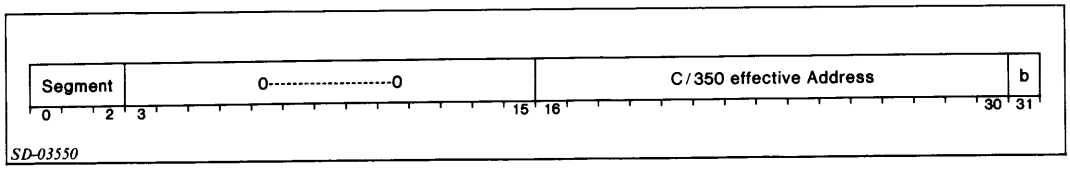


Figure 9.4 C/350 byte addressing format

where

Segment The segment number in bits 0-2 specifies the current segment.

b The byte (b) indicator in bit 31 specifies the byte; b set to one specifies least significant byte (bits 8-15 of a word).

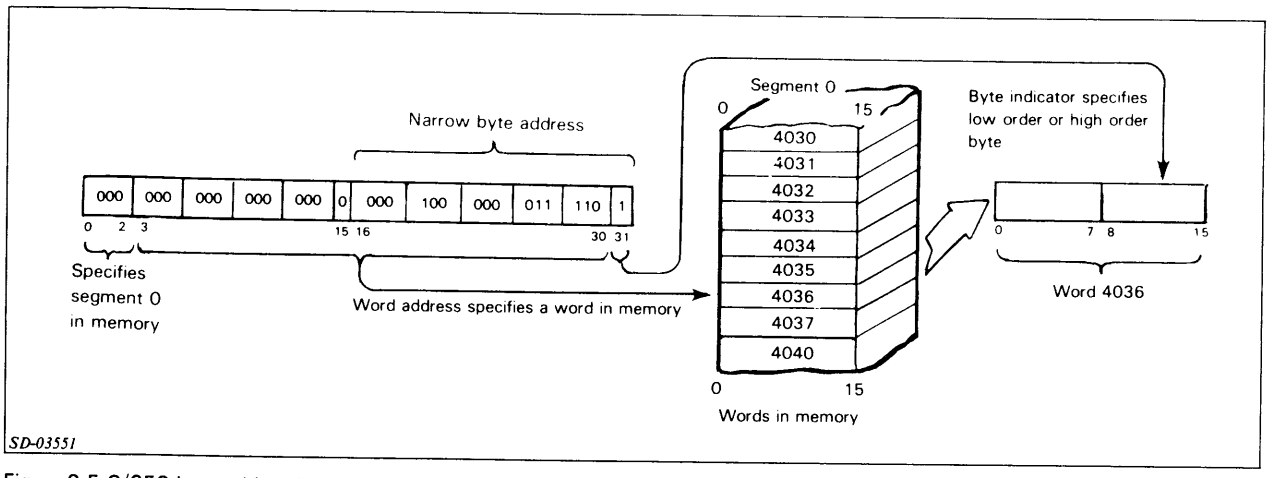


Figure 9.5 C/350 byte addressing

To refer to a bit with a C/350 memory reference instruction (BTO, BTZ, SNB, SZB, and SZBO), the processor forms a bit pointer from the contents of two accumulators. The bit pointer is composed of a word pointer and a bit identifier. The word pointer consists of an effective address (in the ACS accumulator) and a word offset (in the ACD accumulator). The bit identifier is located in the least significant bits of the ACD accumulator.

Figure 9.6 shows the accumulator formats for the BTO, BTZ, SNB, SZB, and SZBO instructions.

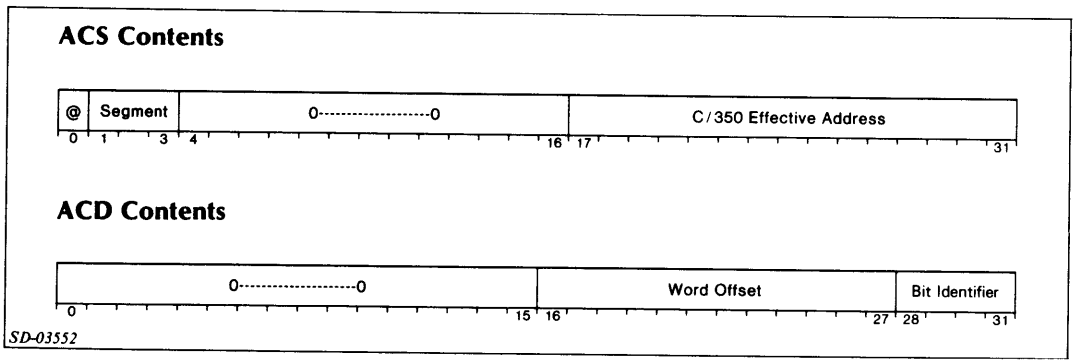


Figure 9.6 C/350 bit addressing format

where

@ The indirect (@) bit in bit 0 of the ACS accumulator forces indirect addressing (when set to one), through a single word pointer.

Segment The segment number specifies the current segment.

The processor uses the ACS accumulator contents to calculate the effective address. For the BTO and BTZ instructions, the processor limits effective addressing to the first 64 Kbytes of the current segment. If a bit instruction specifies the two accumulators as the same accumulator, then the effective address is zero in the current segment.

In Figure 9.7, notice that the processor adds the word offset, an unsigned integer, to the effective address and arrives at a final word address. The processor then locates the bit using the bit identifier, which specifies the bit position (0-15) in the final word.

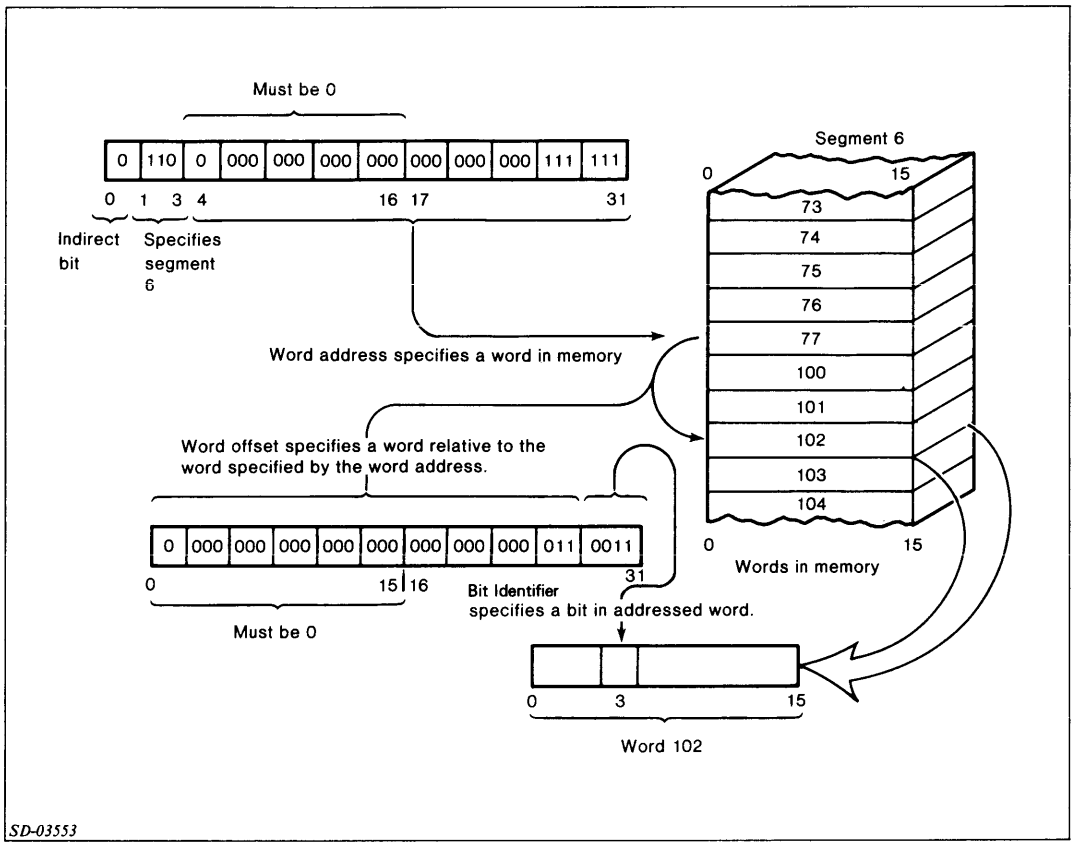


Figure 9.7 BTO, BTZ, SNB, SZB, and SZBO bit addressing

Fixed-Point Instructions

Table 9.2 lists the C/350 fixed-point instructions that refer to memory. The table also shows an equivalent 32-bit processor instruction that you can substitute to expand (within the segment) the memory address range.

Unless otherwise stated, the C/350 instruction and the 32-bit processor equivalent instruction use identical

- Single or double word instruction length
- Argument string
- Data access for writing and for reading (register or memory)
- Data precision of 16 bits

However, an equivalent 32-bit processor instruction uses a double word indirect pointer, while the C/350 instruction uses a single word indirect pointer.

C/350 Instruction	Operation	Equivalent Instruction
BAM	Block add and move	-
BLM	Block move	**WBLM**
BTO	Set bit to one	**WBTO**
BTZ	Set bit to zero	**WBTZ**
CLM	Compare to limits and skip	**WCLM**
CMP	Character compare	**WCMP**
CMT	Character move until true	**WCMT**
CMV	Character move	**WCMV**
COB	Count bits	**WCOB**
CTR	Character translate and compare	**WCTR**
DSZ	Decrement and skip if zero	**XNDSZ** *
EDIT	Edit decimal and alphanumeric 16-bit data	**WEDIT**
EDSZ	Extended decrement and skip if zero	**XNDSZ**
EISZ	Extended increment and skip if zero	**XNISZ**
ELDA	Extended load accumulator	**XNLDA**
ELDB	Extended load byte (from memory to AC)	**XLDB**
ESTA	Extended store accumulator	**XNSTA**
ESTB	Extended store byte (right byte of AC to byte in memory)	**XSTB**
ISZ	Increment and skip if zero	**XNISZ** *
LDA	Load accumulator	**XNLDA** *
LDB	Load byte (from memory to AC)	**WLDB**
LSN	Load sign	**WLSN**
POP	Pop multiple accumulators	**WPOP**
PSH	Push multiple accumulators	**WPSH**
SNB	Skip on nonzero bit	**WSNB**
SZB	Skip on zero bit	**WSZB**
SZBO	Skip on zero bit and set to one	**WSZBO**
STA	Store accumulator	**XNSTA** *
STB	Store byte (right byte of AC to byte in memory)	**WSTB**

Table 9.2 C/350 Fixed-point computing instructions

*The 32-bit processor equivalent instruction requires two words.

Floating-Point Instructions

Table 9.3 lists the C/350 floating-point instructions that refer to memory. The table also shows an equivalent 32-bit processor instruction that you can substitute to expand (within the segment) the memory address range.

Unless otherwise stated, the C/350 instruction and the 32-bit processor equivalent instruction use identical

- Single or double word instruction length
- Argument string
- Data access for writing and for reading (register or memory)
- Data precision of 16, 32, or 64 bits

However, an equivalent 32-bit processor instruction uses a double word indirect pointer, while the C/350 instruction uses a single word indirect pointer.

C/350 Instruction	Operation	Equivalent Instruction
FAMD	Add double (memory to FPAC)	**XFAMD**
FAMS	Add single (memory to FPAC)	**XFAMS**
FDMD	Divide double (FPAC by memory)	**XFDMD**
FDMS	Divide single (FPAC by memory)	**XFDMS**
FFMD	Fix to memory (FPAC to memory)	**WFFAD ***
FLDD	Load floating-point double	**XFLDD**
FLDS	Load floating-point single	**XFLDS**
FLMD	Float from memory	**WFLAD ***
FLST	Load floating-point status register	**LFLST ****
FMMD	Multiply double (FPAC by memory)	**XFMMD**
FMMS	Multiply single (FPAC by memory)	**XFMMS**
FPOP	Pop floating-point state	**WFPOP**
FPSH	Push floating-point state	**WFPSH**
FSMD	Subtract double (memory from FPAC)	**XFSMD**
FSMS	Subtract single (memory from FPAC)	**XFSMS**
FSST	Store floating-point status register	**LFSST ****
FSTD	Store floating-point double	**XFSTD**
FSTS	Store floating-point single	**XFSTS**
LDI	Load integer (memory to FPAC)	**WLDI**
LDIX	Load integer extended (memory to FPAC)	**WLDIX**
STI	Store integer (FPAC to memory)	**WSTI**
STIX	Store integer extended (FPAC to memory)	**WSTIX**

Table 9.3 C/350 Floating-point computing instructions

*The WFFAD and WFLAD instruction use a 32-bit accumulator, while the equivalent C/350 instruction uses two memory words.

**The LFLST or LFSST instruction is a triple word instruction, while the C/350 instruction is a double word instruction.

Chapter 10
Instruction Dictionary

The Instruction Dictionary presents the global 16- and 32-bit instructions. The instructions appear in alphabetical order of the Assembler instruction mnemonic. Each instruction description includes

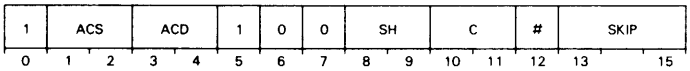
- Assembler instruction mnemonic and statement format.
- Bit format for the Assembler statement.
- Functional description of each instruction.

The Instruction Dictionary uses the following conventions and abbreviations.

UPPERCASE and/or **boldface**	Uppercase or boldface characters indicate a literal argument in an Assembler statement. When you include a literal argument with an Assembler statement, use the exact form.
lowercase and/or *italic*	Lowercase or italic characters indicate a variable argument in an Assembler statement. When you include the argument with an Assembler statement, substitute a literal value for the variable argument.
[]	The square brackets indicate an optional argument. Omit the square brackets when you include an optional argument with an Assembler statement.
ac	Accumulator
acs	Source accumulator
acd	Destination accumulator
fpac	Floating-point accumulator
fpacs	Floating-point source accumulator
fpacd	Floating-point destination accumulator

Add Complement

ADC*[c][sh][#] acs,acd[,skip]*



Adds the logical complement of an unsigned integer to another unsigned integer.

Initializes carry to the specified value; adds the logical complement of the unsigned, 16-bit number in bits 16–31 of ACS to the unsigned, 16-bit number in bits 16–31 of ACD; and places the result in the shifter. The instruction then performs the specified shift operation and loads the result of the shift into bits 16–31 of ACD if the no-load bit is 0. If the skip condition is true, the next sequential word is skipped. For this instruction, *overflow* is 0.

If the load option is specified, bits 0–15 of ACD are undefined.

> **NOTE:** *If the sum of the two numbers being added is greater than 65,535, the instruction complements carry.*

[c]

The processor determines the effect of the CARRY flag (c) on the old value of CARRY before performing the operation (opcode). The following list gives the values of c, bits 10 and 11, and the operation.

Symbol *[c]*	Bits 10-11	Operation
omitted	0 0	Leave CARRY unchanged
Z	0 1	Initialize CARRY to 0
O	1 0	Initialize CARRY to 1
C	1 1	Complement CARRY

[sh]

The processor shifts the CARRY flag and the 16 data bits after performing the instruction operation. The processor can shift the bits left or right one bit position, or it can swap the two bytes. The following list gives the values of sh, bits 8 and 9, and the shift operation.

Symbol *[sh]*	Bits 8-9	Shift Operation
omitted	0 0	Do not shift the result
L	0 1	Shift left
R	1 0	Shift right
S	1 1	Swap the two 8-bit bytes

[#]

Unless you use the no-load option (#), the processor loads the result of the shift operation into the destination accumulator. The no-load option is useful to test the result of the instruction operation without destroying the destination accumulator contents. The following list gives the values of the no-load option, bit 12, and the operation.

Symbol *[#]*	Bit 12	Operation
omitted	0	Load the result into ACD
#	1	Do not load the result and restore the CARRY flag

NOTE: *Do not specify an instruction with the no-load option (#) in combination with either the never skip or always skip option. Thus, the instruction may not end in 1000_2 or 1001_2, other instructions use the bit combinations.*

[skip]

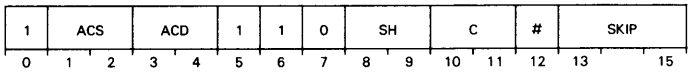
The processor can skip the next instruction if the condition test is true. The following lists gives the test conditions, bits 13 to 15, and the operation.

Symbol *[skip]*	Bits 13-15	Operation
omitted	0 0 0	Never skip
SKP	0 0 1	Always skip
SZC	0 1 0	Skip if CARRY is 0
SNC	0 1 1	Skip if CARRY is not 0
SZR	1 0 0	Skip if the result is 0
SNR	1 0 1	Skip if the result is not 0
SEZ	1 1 0	Skip if either CARRY or the result is 0
SBN	1 1 1	Skip if both CARRY and the result are not 0

When the instruction performs a skip, it skips the next sequential 16-bit word. Make sure that a skip does not transfer control to the middle of a 32-bit or longer instruction.

Add

ADD*[c][sh][#] acs,acd[,skip]*



Performs unsigned integer addition and complements carry if appropriate.

Initializes carry to the specified value; adds the unsigned, 16-bit number in bits 16–31 of ACS to the unsigned, 16-bit number in bits 16–31 of ACD; and places the result in the shifter. The instruction then performs the specified shift operation and places the result of the shift in bits 16–31 of ACD if the no-load bit is 0. If the skip condition is true, the next sequential word is skipped. For this instruction, *overflow* is 0.

If the load option is specified, bits 0–15 of ACD are undefined.

NOTE: *If the sum of the two numbers being added is greater than 65,535, the instruction complements carry.*

[c]

The processor determines the effect of the CARRY flag (c) on the old value of CARRY before performing the operation (opcode). The following list gives the values of c, bits 10 and 11, and the operation.

Symbol *[c]*	Bits 10-11	Operation
omitted	0 0	Leave CARRY unchanged
Z	0 1	Initialize CARRY to 0
O	1 0	Initialize CARRY to 1
C	1 1	Complement CARRY

[sh]

The processor shifts the CARRY flag and the 16 data bits after performing the instruction operation. The processor can shift the bits left or right one bit position, or it can swap the two bytes. The following list gives the values of sh, bits 8 and 9, and the shift operation.

Symbol *[sh]*	Bits 8-9	Shift Operation
omitted	0 0	Do not shift the result
L	0 1	Shift left
R	1 0	Shift right
S	1 1	Swap the two 8-bit bytes

[#]

Unless you use the no-load option (#), the processor loads the result of the shift operation into the destination accumulator. The no-load option is useful to test the result of the instruction operation without destroying the destination accumulator contents. The following list gives the values of the no-load option, bit 12, and the operation.

Symbol *[#]*	Bit 12	Operation
omitted	0	Load the result into ACD
#	1	Do not load the result and restore the CARRY flag

NOTE: *Do not specify an instruction with the no-load option (#) in combination with either the never skip or always skip option. Thus, the instruction may not end in 1000_2 or 1001_2, other instructions use the bit combinations.*

[skip]

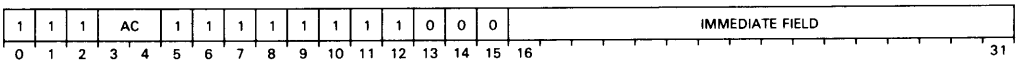
The processor can skip the next instruction if the condition text is true. The following lists gives the test conditions, bits 13 to 15, and the operation.

Symbol *[skip]*	Bits 13-15	Operation
omitted	0 0 0	Never skip
SKP	0 0 1	Always skip
SZC	0 1 0	Skip if CARRY is 0
SNC	0 1 1	Skip if CARRY is not 0
SZR	1 0 0	Skip if the result is 0
SNR	1 0 1	Skip if the result is not 0
SEZ	1 1 0	Skip if either CARRY or the result is 0
SBN	1 1 1	Skip if both CARRY and the result are not 0

When the instruction performs a skip, it skips the next sequential 16-bit word. Make sure that a skip does not transfer control to the middle of a 32-bit or longer instruction.

Extended Add Immediate

ADDI *i,ac*



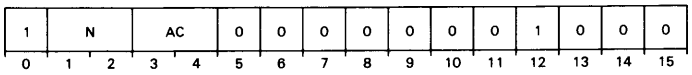
Adds a signed integer in the range of −32,768 to +32,767 to the contents of an accumulator.

Treats the contents of the immediate field as a signed, 16-bit, two's complement number and adds it to the signed, 16-bit, two's complement number contained in bits 16–31 of the specified accumulator, placing the result in bits 16–31 of the same accumulator. Carry remains unchanged and *overflow* is 0.

Bits 0–15 of the modified accumulator are undefined after completion of this instruction.

Add Immediate

ADI *n,ac*



Adds an unsigned integer in the range 1–4 to the contents of an accumulator.

Adds the contents of the immediate field *N*, plus 1, to the unsigned, 16-bit number contained in bits 16–31 of the specified accumulator, placing the result in bits 16–31 of the same accumulator. Carry remains unchanged and *overflow* is 0.

Bits 0–15 of the modified accumulator are undefined after completion of this instruction.

NOTE: *The assembler takes the coded value of* n *and subtracts 1 from it before placing it in the immediate field. Therefore, the programmer should code the exact value that is to be added.*

Example

Assume that AC2 contains 177775_8. After the instruction **ADI 4,2** is executed, AC2 contains 000001_8 and carry is unchanged. (See Figure 10.1.)

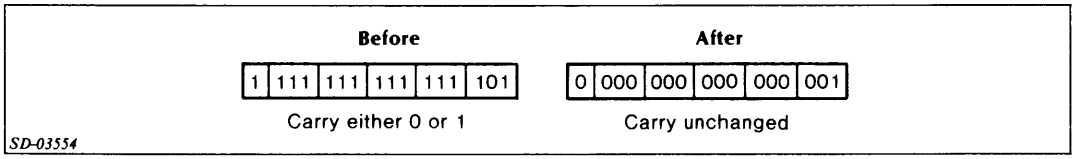
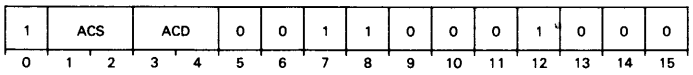


Figure 10.1 ADI example

AND with Complemented Source

ANC *acs,acd*



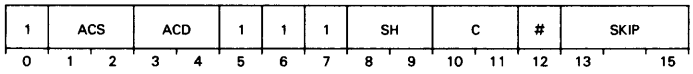
ANDs the contents of an accumulator with the logical complement of another accumulator.

Forms the logical AND of the logical complement of the contents of bits 16–31 of ACS and the contents of bits 16–31 of ACD and places the result in bits 16–31 of ACD. The instruction sets a bit position in the result to 1 if the corresponding bits in ACS contain 0 and ACD contain 1. The contents of carry and ACS remain unchanged. *Overflow* is 0.

Bits 0–15 of the modified accumulator are undefined after completion of this instruction.

AND

AND*[c][sh][#] acs,acd[,skip]*



Forms the logical AND of the contents of two accumulators.

Initializes carry to the specified value. Places the logical AND of bits 16–31 of ACS and bits 16–31 of ACD in the shifter. Each bit placed in the shifter is 1 only if the corresponding bit in both ACS and ACD is one; otherwise the resulting bit is 0. The instruction then performs the specified shift operation and places the result in bits 16–31 of ACD if the no-load bit is 0. If the skip condition is true, the next sequential word is skipped. *Overflow* is 0.

If the load option is specified, bits 0–15 of ACD are undefined.

[c]

The processor determines the effect of the CARRY flag (c) on the old value of CARRY before performing the operation (opcode). The following list gives the values of c, bits 10 and 11, and the operation.

Symbol *[c]*	Bits 10-11	Operation
omitted	0 0	Leave CARRY unchanged
Z	0 1	Initialize CARRY to 0
O	1 0	Initialize CARRY to 1
C	1 1	Complement CARRY

[sh]

The processor shifts the CARRY flag and the 16 data bits after performing the instruction operation. The processor can shift the bits left or right one bit position, or it can swap the two bytes. The following list gives the values of sh, bits 8 and 9, and the shift operation.

Symbol *[sh]*	Bits 8-9	Shift Operation
omitted	0 0	Do not shift the result
L	0 1	Shift left
R	1 0	Shift right
S	1 1	Swap the two 8-bit bytes

[#]

Unless you use the no-load option (#), the processor laods the result of the shift operation into the destination accumulator. The no-load option is useful to test the result of the instruction operation without destroying the destination accumulator contents. The following list gives the values fo the no-load option, bit 12, and the operation.

Symbol *[#]*	Bit 12	Operation
omitted	0	Load the result into ACD
#	1	Do not load the result and restore the CARRY flag

NOTE: *Do not specify an instruction with the no-load option (#) in combination with either the never skip or always skip option. Thus, the instruction may not end in 1000_2 or 1001_2, other instructions use the bit combinations.*

[skip]

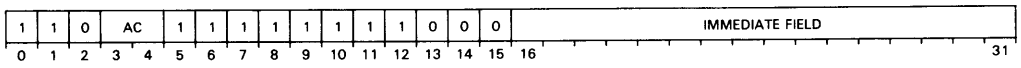
The processor can skip the next instruction if the condition test is true. The following list gives the test conditions, bits 13 to 15, and the operation.

Symbol *[skip]*	Bits 13-15	Operation
omitted	0 0 0	Never skip
SKP	0 0 1	Always skip
SZC	0 1 0	Skip if CARRY is 0
SNC	0 1 1	Skip if CARRY is not 0
SZR	1 0 0	Skip if the result is 0
SNR	1 0 1	Skip if the result is not 0
SEZ	1 1 0	Skip if either CARRY or the result is 0
SBN	1 1 1	Skip if both CARRY and the result are not 0

When the instruction performs a skip, it skips the next sequential 16-bit word. Make sure that a skip does not transfer control to the middle of a 32-bit or longer instruction.

AND Immediate

ANDI *i,ac*



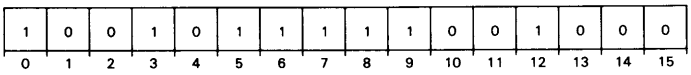
ANDs the contents of an accumulator with the contents of a 16-bit number contained in the instruction.

Places the logical AND of the contents of the immediate field and the contents of bits 16–31 of the specified accumulator in bits 16–31 of the specified accumulator. Carry is unchanged and *overflow* is 0.

Bits 0–15 of the modified accumulator are undefined after completion of this instruction.

Block Add and Move

BAM



Moves memory words from one location to another, adding a constant to each one.

Moves words sequentially from one memory location to another, treating them as unsigned, 16-bit integers. After fetching a word from the source location, the instruction adds the unsigned, 16-bit integer in bits 16–31 of AC0 to it. If the addition produces a result that is greater than 32,768, no indication is given.

Bits 17–31 of AC2 contain the address of the source location. Bits 17–31 of AC3 contain the address of the destination location. The address in bits 17–31 of AC2 or AC3 is an indirect address if bit 16 of that accumulator is 1. In that case, the instruction follows the indirection chain before placing the resultant effective address in the accumulator.

The unsigned, 16-bit number in bits 16–31 of AC1 is equal to the number of words moved. This number must be greater than 0 and less than or equal to 32,768. If the number in AC1 is outside these bounds, no data is moved and the contents of the accumulators remain unchanged.

AC	Contents
0	Addend
1	Number of words to be moved
2	Source address
3	Destination address

For each word moved, the count in AC1 is decremented by one and the source and destination addresses in AC2 and AC3 are incremented by one. Upon completion of the instruction, AC1 contains zeroes, and AC2 and AC3 point to the word following the last word in their respective fields. The contents of carry and AC0 remain unchanged. *Overflow* is 0.

The 31-bit effective address generated by this instruction is constrained to be within the first 64 Kbytes of the current segment.

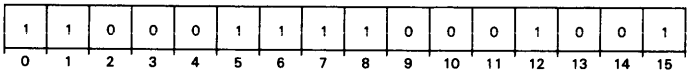
Words are moved in consecutive, ascending order according to their addresses. The next address after 77777_8 is 0 for both fields. The fields may overlap in any way.

> **NOTE:** *Because of the potentially long time that may be required to perform this instruction, it is interruptible. If a* Block Add and Move *instruction is interrupted, the program counter is decremented by one before it is placed in location 0 so that it points to the interrupted instruction. Because the addresses and the word count are updated after every word stored, any interrupt service routine that returns control to the interrupted program via the address stored in memory location 0 will correctly restart the* Block Add and Move *instruction.*

When updating the source and destination addresses, the *Block Add and Move* instruction forces bit 0 of the result to 0. This ensures that upon return from an interrupt, the *Block Add and Move* instruction will not try to resolve an indirect address in either AC2 or AC3.

Breakpoint

BKPT



The *Breakpoint* **BKPT** instruction pushes a wide return block onto the stack and transfers program control to the breakpoint handler.

Before executing the **BKPT** instruction, you must first store, in memory, the one-word opcode from the location that the **BKPT** instruction will occupy. Then, store the **BKPT** instruction in that one-word location.

When the processor executes the **BKPT** instruction, it pushes a wide return block onto the current stack and jumps to the breakpoint handler. (The value of the PC in the return block is the address of the **BKPT** instruction.)

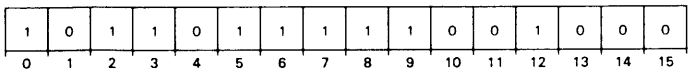
After pushing the return block, the **BKPT** instruction sets the PSR to zero. It also stores, in the PC, the effective address of the wide jump indirect through the breakpoint handler (located in the current segment). Finally, it checks for a stack overflow. If no overflow occurs, control transfers to the breakpoint handler. If stack overflow occurs, the processor services the stack fault (AC1 contains the code 0).

> **NOTE:** *If you remove the BKPT instruction before returning from the breakpoint handler, return with the WPOPB instruction; otherwise use the PBX instruction.*

Carry and *overflow* are indeterminate as a result of executing the **BKPT** instruction.

Block Move

BLM



Moves memory words from one location to another.

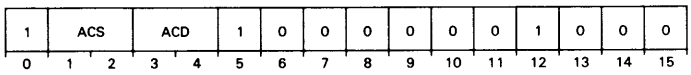
The *Block Move* instruction is the same as the *Block Add and Move* instruction in all respects except that no addition is performed and AC0 is not used. Carry remains unchanged and *overflow* is 0.

The 31-bit effective address generated by this instruction is constrained to be within the first 64 Kbytes of the current segment.

NOTE: *The* Block Move *instruction is interruptible in the same manner as the* Block Add and Move *instruction.*

Set Bit to One

BTO *acs,acd*



Sets the specified bit to 1.

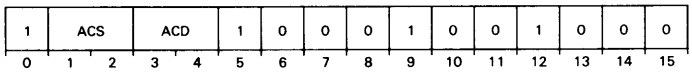
Forms a 32-bit bit pointer from the contents of bits 16–31 of both ACS and ACD. Bits 16–31 of ACS contain the high-order 16 bits and bits 16–31 of ACD contain the low-order 16 bits of the bit pointer. If ACS and ACD are specified as the same accumulator, the instruction treats the accumulator contents as the low-order 16-bits of the bit pointer and assumes the high-order 16 bits are 0. Carry remains unchanged and *overflow* is 0.

The instruction then sets the addressed bit in memory to 1, leaving the contents of ACS and ACD unchanged.

The 31-bit effective address generated by this instruction is constrained to be within the first 64 Kbytes of the current segment.

Set Bit to Zero

BTZ *acs,acd*



Sets the addressed bit to 0.

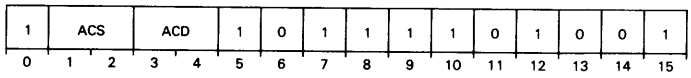
Forms a 32-bit bit pointer from the contents of bits 16–31 of both ACS and ACD. Bits 16–31 of ACS contains the high-order 16 bits and bits 16–31 of ACD contains the low-order 16 bits of the bit pointer. If ACS and ACD are specified as the same accumulator, the instruction treats the accumulator contents as the low-order 16 bits of the bit pointer and assumes the high-order 16 bits are 0. Carry remains unchanged and *overflow* is 0.

The instruction then sets the addressed bit in memory to 0, leaving the contents of ACS and ACD unchanged.

The 31-bit effective address generated by this instruction is constrained to be within the first 64 Kbytes of the current segment.

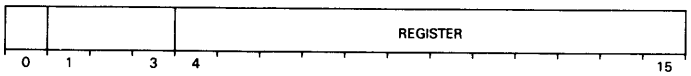
Command I/O

CIO *acs,acd*



Issues a read or write data command using the I/O channel bus. Carry is unchanged and *overflow* is 0.

The command must have the form:



Bits 16–31 of ACS contain the command. Bit 16 of ACS indicates whether a read or a write operation is to take place.

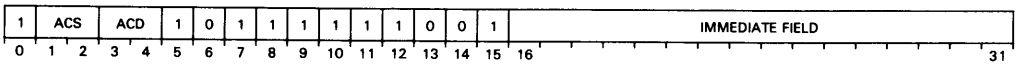
Bits 17–19 specify the I/O channel to use. I/O channel numbers range from 0–7.

The instruction issues the command contained in ACS directly via the I/O channel bus. Bit 16 of ACS determines the operation to perform. If bit 16 of ACS is 0, the instruction performs a read data operation. The instruction receives the data via the I/O channel bus and loads it into bits 16–31 of ACD. Bits 0–15 of ACD are undefined.

If bit 16 of ACS is 1, the instruction performs a write data operation and sends the data in bits 16–31 of ACD via the I/O channel bus.

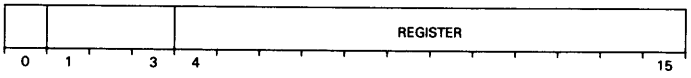
Command I/O Immediate

CIOI *i,acs,acd*



Issues a command via the I/O channel bus. Carry is unchanged and *overflow* is 0.

The command must have the form:



If ACS and ACD are the same, then the immediate field contains the command to be issued on the I/O channel bus (bits 17–19).

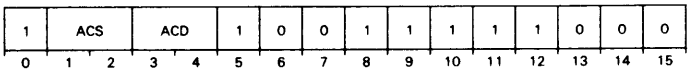
If ACS and ACD are different, then the logical OR of the immediate field and bits 16–31 of ACS is the command to be issued on the I/O channel bus.

If bit 16 of the command is a 0, then a read data operation issued via the I/O channel bus loads the received data into bits 16–31 of ACD. Bits 0–15 of ACD remain undefined.

If bit 16 of this state is 1, then a write data operation issued via the I/O channel bus sends the contents of ACD bits 16–31 to the device.

Compare to Limits

CLM *acs,acd*



Compares a signed integer with two other integers and skips if the first integer is between the other two. The accumulators determine the location of the three integers.

Compares the 16-bit, signed, two's complement integer in bits 16–31 of ACS to two 16-bit, signed, two's complement limit values, *L* and *H*. If the number in bits 16–31 of ACS is greater than or equal to *L* and less than or equal to *H*, the next sequential word is skipped. If the number in bits 16–31 of ACS is less than *L* or greater than *H*, the next sequential word is executed.

If ACS and ACD are specified as different accumulators, the address of the limit value *L* is contained in bits 17–31 of ACD. The limit value *H* is contained in the word following *L*. Bits 0–16 of ACD are ignored.

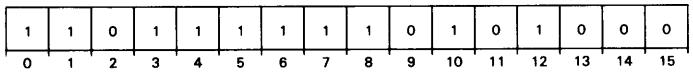
The 31-bit effective address generated by this instruction is constrained to be within the first 64 Kbytes of the current segment.

If ACS and ACD are specified as the same accumulator, then the integer to be compared must be in that accumulator and the limit values *L* and *H* must be in the two words following the instruction. *L* is the first word and *H* is the second word. The next instruction to be executed will begin with the third word following the instruction.

When *L* and *H* are in line, this instruction can be placed anywhere in the 32-bit address space. The instruction leaves carry unchanged; *overflow* is 0.

Character Compare

CMP



Under control of the four accumulators, compares two strings of bytes and returns a code in AC1 reflecting the results of the comparison.

The instruction compares the strings one byte at a time. Each byte is treated as an unsigned 8-bit binary quantity in the range $0–255_{10}$. If two bytes are not equal, the string whose byte has the smaller numerical value is, by definition, the *lower valued* string. Both strings remain unchanged. The four accumulators contain parameters passed to the instruction. Two accumulators specify the starting address, the number of bytes, and the direction of processing (ascending or descending addressed) for each string.

Bits 16–31 of AC0 specify the length and direction of comparison for string 2. If the string is to be compared from its lowest memory location to the highest, bits 16–31 of AC0 contain the unsigned value of the number of bytes in string 2. If the string is to be compared from its highest memory location to the lowest, bits 16–31 of AC0 contain the two's complement of the number of bytes in string 2.

Bits 16–31 of AC1 specify the length and direction of comparison for string 1. If the string is to be compared from its lowest memory location to the highest, bits 16–31 of AC0 contain the unsigned value of the number of bytes in string 1. If the string is to be compared from its highest memory location to the lowest, bits 16–31 of AC1 contain the two's complement of the number of bytes in string 1.

Bits 16–31 of AC2 contain a byte pointer to the first byte compared in string 2. When the string is compared in ascending order, AC2 points to the lowest byte. When the string is compared in descending order, AC2 points to the highest byte.

Bits 16–31 of AC3 contain a byte pointer to the first byte compared in string 1. When the string is compared in ascending order, AC3 points to the lowest byte. When the string is compared in descending order, AC3 points to the highest byte.

Code	Comparison Result
−1	string 1 < string 2
0	string 1 = string 2
+1	string 1 > string 2

The strings may overlap in any way. Overlap will not effect the results of the comparison.

Upon completion, bits 16–31 of AC0 contain the number of bytes (or the two's complement of the number of bytes) left to compare in string 2. AC1 contains the return code as shown in the list above. Bits 16–31 of AC2 contain a byte pointer either to the failing byte in string 2 (if an inequality were found) or to the byte following string 2 (if string 2 were exhausted). Bits 16–31 of AC3 contains a byte pointer either to the failing byte in string 1 (if an inequality were found) or to the byte following string 1 (if string 1 were exhausted). Carry remains unchanged. *Overflow* is 0.

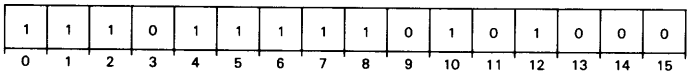
If AC0 and AC1 both contain 0 (both string 1 and string 2 have length zero), the instruction compares no bytes and returns 0 in AC1. If the two strings are of unequal length, the instruction pads the shorter string with space characters $<040_8>$ and continues the comparison.

The 31-bit effective address generated by this instruction is constrained to be within the first 64 Kbytes of the current segment.

NOTE: *The original contents of AC2 and AC3 must be valid byte pointers to an area in the user's address space. If the pointers are invalid, a protection fault occurs, even if no bytes are to be compared. AC1 contains the code 2.*

Character Move Until True

CMT



Under control of the four accumulators, moves a string of bytes from one area of memory to another until either a table-specified delimiter character is encountered or the source string is exhausted.

The instruction copies the string one byte at a time. Before it moves a byte, the instruction uses that byte's value to determine if it is a delimiter. It treats the byte as an unsigned 8-bit binary integer (in the range $0–255_{10}$) and uses it as a bit index into a 256-bit delimiter table. If the indexed bit in the delimiter table is zero, the byte pending is not a delimiter and the instruction copies it from the source string to the destination string. If the indexed bit in the delimiter table is 1, the byte pending is a delimiter; the instruction does not copy it and the instruction terminates.

The instruction processes both strings in the same direction, either from lowest memory locations to highest (*ascending order*), or from highest memory locations to lowest (*descending order*). Processing continues until there is a delimiter or the source string is exhausted. The four accumulators contain parameters passed to the instruction.

Bits 16–31 of AC0 contain the address (word address), possibly indirect, of the start of the 256-bit (16-word) delimiter table.

Bits 16–31 of AC1 specify the length of the strings and the direction of processing. If the source string is to be moved to the destination field in ascending order, bits 16–31 of AC1 contain the unsigned value of the number of bytes in the source string. If the source string is to be moved to the destination field in descending order, bits 16–31 of AC1 contain the two's complement of the number of bytes in the source string.

Bits 16–31 of AC2 contain a byte pointer to the first byte to be written in the destination field. When the process is performed in ascending order, bits 16–31 of AC2 point to the lowest byte in the destination field. When the process is performed in descending order, bits 16–31 of AC2 point to the highest byte in the destination field.

Bits 16–31 of AC3 contain a byte pointer to the first byte to be processed in the source string. When the process is performed in ascending order, bits 16–31 of AC3 point to the lowest byte in the source string. When the process is performed in descending order, bits 16–31 of AC3 point to the highest byte in the source string.

The fields may overlap in any way. However, the instruction moves bytes one at a time, so certain types of overlap may produce unusual side effects.

Upon completion, bits 16–31 of AC0 contain the resolved address of the translation table and AC1 contain the number of bytes (or the two's complement of the number of bytes) that were not moved. Bits 16–31 of AC2 contain a byte pointer to the byte following the last byte written in the destination field. Bits 16–31 of AC3 contain a byte pointer either to the delimiter or to the first byte following the source string. Carry remains unchanged. *Overflow* is 0.

The 31-bit effective address generated by this instruction is constrained to be within the first 64 Kbytes of the current segment.

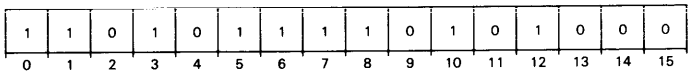
NOTES: *If AC1 contains the number 0 at the beginning of this instruction, no bytes are fetched and none are stored. The instruction becomes a No-Op.*

If AC2=AC3, then no characters are written; the string is scanned for the delimiter.

The original contents of AC0, AC2, and AC3 must be valid pointers to some area in the user's address space. If they are invalid, a protection fault occurs, even if no bytes are to be moved. AC1 contains the code 2.

Character Move

CMV



Under control of the four accumulators, moves a string of bytes from one area of memory to another and returns a value in carry reflecting the relative lengths of source and destination strings.

The instruction copies the source string to the destination field, one byte at a time. The four accumulators contain parameters passed to the instruction. Two accumulators specify the starting address, number of bytes to be copied, and the direction of processing (ascending or descending addresses) for each field.

Bits 16–31 of AC0 specify the length and direction of processing for the destination field. If the field is to be processed from its lowest memory location to the highest, bits 16–31 of AC0 contain the unsigned value of the number of bytes in the destination field. If the field is to be processed from its highest memory location to the lowest, bits 16–31 of AC0 contain the two's complement of the number of bytes in the destination field.

Bits 16–31 of AC1 specify the length and direction of processing for the source string. If the string is to be processed from its lowest memory location to the highest, bits 16–31 of AC1 contain the unsigned value of the number of bytes in the source string. If the field is to be processed from its highest memory location to the lowest, bits 16–31 of AC1 contain the two's complement of the number of bytes in the source string.

Bits 16–31 of AC2 contain a byte pointer to the first byte to be written in the destination field. When the field is written in ascending order, bits 16–31 of AC2 point to the lowest byte. When the field is written in descending order, bits 16–31 of AC2 point to the highest byte.

Bits 16–31 of AC3 contain a byte pointer to the first byte copied in the source string. When the field is copied in ascending order, bits 16–31 of AC3 point to the lowest byte. When the field is copied in descending order, bits 16–31 of AC3 point to the highest byte.

The fields may overlap in any way. However, the instruction moves bytes one at a time, so certain types of overlap may produce unusual side effects.

Upon completion, AC0 contains 0 and bits 16–31 of AC1 contain the number of bytes (or the two's complement of the number of bytes) left to fetch from the source field. Bits 16–31 of AC2 contain a byte pointer to the byte following the destination field; bits 16–31 of AC3 contain a byte pointer to the byte following the last byte fetched from the source field. *Overflow* is 0.

The 31-bit effective address generated by this instruction is constrained to be within the first 64 Kbytes of the current segment.

NOTES: *If AC0 contains the number 0 at the beginning of this instruction, no bytes are fetched and none are stored. If AC1 is 0 at the beginning of this instruction, the destination field is filled with space characters.*

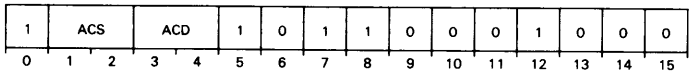
The original contents of AC2 and AC3 must be valid pointers to some area in the user's address space. If they are invalid, a protection fault occurs, even if no bytes are to be moved. AC1 contains the code 2.

If the source field is longer than the destination field, the instruction terminates when the destination field is filled and sets carry to 1. In any other case, the instruction sets carry to 0.

If the source field is shorter than the destination field, the instruction pads the destination field with space characters $<040_8>$.

Count Bits

COB *acs,acd*



Counts and adds the number of ones in an accumulator to another accumulator.

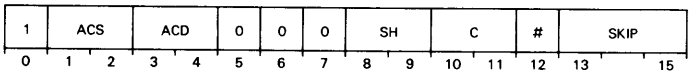
Adds a number equal to the number of ones in bits 16–31 of ACS to the signed, 16-bit, two's complement number in bits 16–31 of ACD. The instruction leaves the contents of ACS and the state of carry unchanged. *Overflow* is 0.

Bits 0–15 of the modified accumulator are undefined after completion of this instruction.

NOTE: *If ACS and ACD are specified to be the same accumulator, the instruction functions as described above, except the contents of ACS will be changed.*

Complement

COM*[c][sh][#] acs,acd[,skip]*



Forms the logical complement of the contents of an accumulator.

Initializes carry to the specified value, forms the logical complement of the number in bits 16–31 of ACS, and performs the specified shift operation. The instruction then places the result in bits 16–31 of ACD if the no-load bit is 0. If the skip condition is true, the next sequential word is skipped.

If the load option is specified, bits 0–15 of ACD are undefined.

For this instruction, *overflow* is 0.

[c]

The processor determines the effect of the CARRY flag (c) on the old value of CARRY before performing the operation (opcode). The following list gives the values of c, bits 10 and 11, and the operation.

Symbol *[c]*	Bits 10-11	Operation
omitted	0 0	Leave CARRY unchanged
Z	0 1	Initialize CARRY to 0
O	1 0	Initialize CARRY to 1
C	1 1	Complement CARRY

[sh]

The processor shifts the CARRY flag and the 16 data bits after performing the instruction operation. The processor can shift the bits left or right one bit position, or it can swap the two bytes. The following list gives the values of sh, bits 8 and 9, and the shift operation.

Symbol *[sh]*	Bits 8-9	Shift Operation
omitted	0 0	Do not shift the result
L	0 1	Shift left
R	1 0	Shift right
S	1 1	Swap the two 8-bit bytes

[#]

Unless you use the no-load option (#), the processor loads the result of the shift operation into the destination accumulator. The no-load option is useful to test the result of the instruction operation without destroying the destination accumulator contents. The following list gives the values of the no-load option, bit 12, and the operation.

Symbol *[#]*	Bit 12	Operation
omitted	0	Load the result into ADC
#	1	Do not load the result and restore the CARRY flag

NOTE: *Do not specify an instruction with the no-load option (#) in combination with either the never skip or always option. Thus, the instruction may not end in 1000_2 or 1001_2, other instructions use the bit combinations.*

[skip]

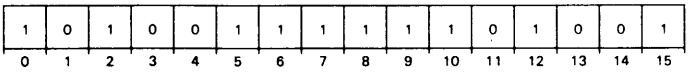
The processor can skip the next instruction if the condition test is true. The following list gives the test conditions, bits 13 to 15, and the operation.

Symbol *[skip]*	Bits 13-15	Operation
omitted	0 0 0	Never skip
SKP	0 0 1	Always skip
SZC	0 1 0	Skip if CARRY is 0
SNC	0 1 1	Skip if CARRY is not 0
SZR	1 0 0	Skip if the result is 0
SNR	1 0 1	Skip if the result is not 0
SEZ	1 1 0	Skip if either CARRY or the result is 0
SBN	1 1 1	Skip if both CARRY and the result are not 0

When the instruction performs a skip, it skips the next sequential 16-bit word. Make sure that a skip does not transfer control to the middle of a 32-bit or longer instruction.

Complement Carry

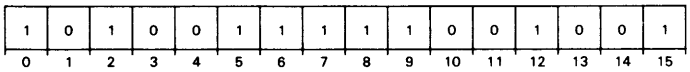
CRYTC



Complements the value of carry. *Overflow* is 0.

Set Carry to One

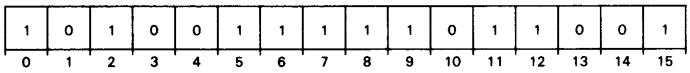
CRYTO



Unconditionally sets the value of carry to 1. *Overflow* is 0.

Set Carry to Zero

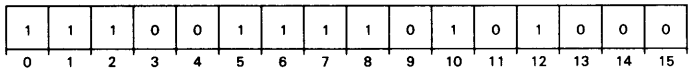
CRYTZ



Unconditionally sets the value of carry to 0. *Overflow* is 0.

Character Translate

CTR



Under control of the four accumulators, translates a string of bytes from one data representation to another and either moves it to another area of memory or compares it to a second translated string.

The instruction operates in two modes: translate and move, and translate and compare.

When operating in translate and move mode, the instruction translates each byte in string 1 and places it in a corresponding position in string 2. Translation is performed by using each byte as an 8-bit index into a 256-byte translation table. The byte addressed by the index then becomes the translated value.

When operating in translate and compare mode, the instruction translates each byte in string 1 and string 2 as described above and compares the translated values. Each translated byte is treated as an unsigned 8-bit binary quantity in the range $0–255_{10}$. If two translated bytes are not equal, the string whose byte has the smaller numerical value is, by definition, the *lower valued* string. Both strings remain unchanged.

Bits 16–31 of AC0 specify the address, either direct or indirect, of a word which contains a byte pointer to the first byte in the 256-byte translation table.

Bits 16–31 of AC1 specify the length of the two strings and the mode of processing. If string 1 is to be processed in translate and move mode, bits 16–31 of AC1 contain the two's complement of the number of bytes in the strings. If the strings are to be processed in translate and compare mode, bits 16–31 of AC1 contain the unsigned value of the number of bytes in the strings. Both strings are processed from lowest memory address to highest.

Bits 16–31 of AC2 contain a byte pointer to the first byte in string 2.

Bits 16–31 of AC3 contain a byte pointer to the first byte in string 1.

Upon completion of a translate and move operation, bits 16–31 of AC0 contain the address of the word which contains the byte pointer to the translation table and AC1 contains 0. Bits 16–31 of AC2 contain a byte pointer to the byte following string 2 and bits 16–31 of AC3 contain a byte pointer to the byte following string 1. Carry remains unchanged. *Overflow* is 0.

Upon completion of a translate and compare operation, bits 16–31 of AC0 contain the address of the word which contains the byte pointer to the translation table. AC1 contains a return code as calculated in the list below. Bits 16–31 of AC2 contain a byte pointer to either the failing byte in string 2 (if an inequality was found) or the byte following string 2 (if the strings were identical). Bits 16–31 of AC3 contain a byte pointer to either the failing byte in string 1 (if an inequality was found) or the byte following string 1 if the strings were identical. Carry contains an indeterminate value. *Overflow* is 0.

The 31-bit effective address generated by this instruction is constrained to be within the first 64 Kbytes of the current segment.

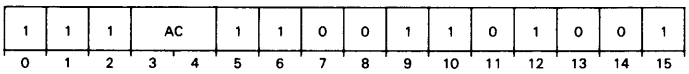
Code	Result
−1	Translated value of string 1 < translated value of string 2
0	Translated value of string 1 = translated value of string 2
+1	Translated value of string 1 > translated value of string 2

If the length of both string 1 and string 2 is 0, the compare option returns a 0 in AC1.

The fields may overlap in any way. However, processing is done one character at a time, so unusual side effects may be produced by certain types of overlap.

Convert to 16-Bit Integer

CVWN *ac*

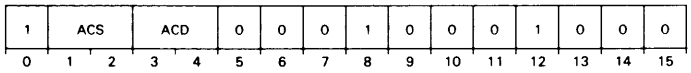


Converts a 32-bit integer to a 16-bit integer.

The instruction converts the 32-bit contents of the specified accumulator to a 16-bit integer by extending bit 16 into bits 0–15. If the 17 most significant bits do not contain the same value (i.e., all 1's or all 0's) before conversion takes place, then this instruction sets *overflow* to 1 before performing the conversion. Carry is unchanged.

Decimal Add

DAD *acs,acd*



Performs decimal addition on 4-bit binary coded decimal (BCD) numbers and uses carry for a decimal carry.

Adds the unsigned decimal digit contained in bits 28–31 of ACS to the unsigned decimal digit contained in bits 28–31 of ACD. Carry is added to this result. The instruction then places the decimal units' position of the final result in bits 28–31 of ACD and the decimal carry in carry. The contents of ACS and bits 0–27 of ACD remain unchanged. *Overflow* is 0.

> **NOTE:** *No validation of the input digits is performed. Therefore, if bits 28–31 of either ACS or ACD contain a number greater than 9, the results will be unpredictable.*

Example

Assume that bits 28–31 of AC2 contain 9, bits 28–31 of AC3 contain 7, and carry is 0. After the instruction **DAD 2,3** is executed, AC2 remains the same; bits 28–31 of AC3 contain 6; and carry is 1, indicating a decimal carry from this *Decimal Add*. (See Figure 10.2.)

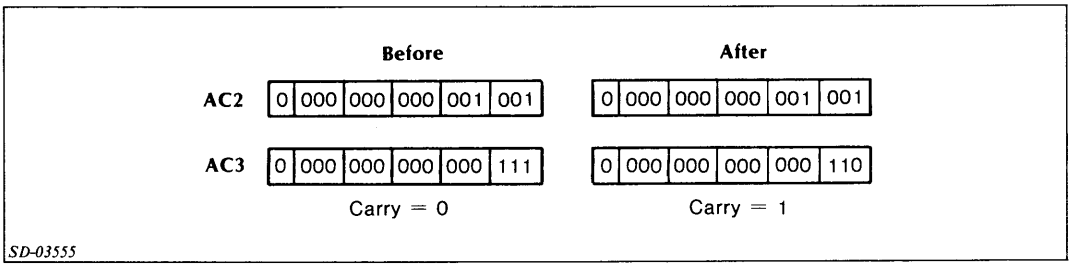
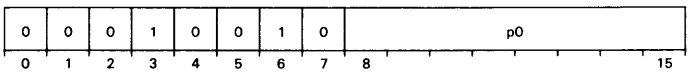


Figure 10.2 DAD example

Add to DI (edit subopcode)

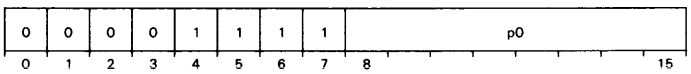
DADI *p0*



Adds the 8-bit two's complement integer specified by *p0* to the Destination Indicator (DI).

Add to P Depending on S (edit subopcode)

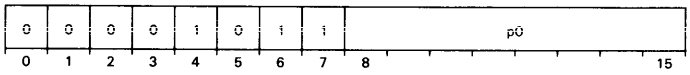
DAPS *p0*



If *S* is 0, the instruction adds the 8-bit two's complement integer specified by *p0* to the opcode Pointer (P). Before the add is performed, P is pointing to the byte containing the **DAPS** opcode.

Add to P Depending on T (edit subopcode)

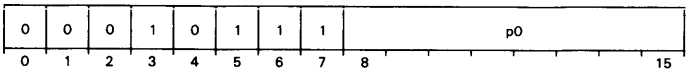
DAPT *p0*



If *T* is one, the instruction adds the 8-bit two's complement integer specified by *p0* to the opcode Pointer (P). Before the add is performed, P is pointing to the byte containing the **DAPT** opcode.

Add to P (edit subopcode)

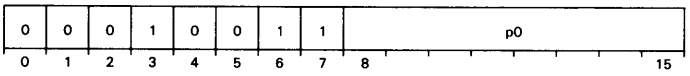
DAPU *p0*



Adds the 8-bit two's complement integer specified by *p0* to the opcode Pointer (P). Before the add is performed, P is pointing to the byte containing the **DAPU** opcode.

Add to SI (edit subopcode)

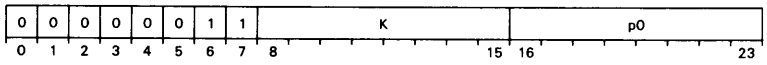
DASI *p0*



Adds the 8-bit two's complement integer specified by *p0* to the Source Indicator (SI).

Decrement and Jump if Nonzero (edit subopcode)

DDTK *k,p0*



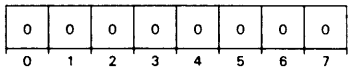
Decrements a word in the stack by one. If the decremented value of the word is nonzero, the instruction adds the 8-bit two's complement integer specified by *p0* to the opcode Pointer (P). Before the add is performed, P is pointing to the byte containing the **DDTK** opcode.

For EDIT if the 8-bit two's complement integer specified by *k* is negative, the word decremented is at the address, *narrow stack pointer*+1+*k*. If *k* is positive, the word decremented is at the address, *narrow frame pointer*+1+*k*.

For WEDIT if the 8-bit two's complement integer specified by *k* is negative, the word decrement is at address WSP + 2 + (2*)*k*. If *k* is positive, the word decremented is at address WFP + 2 + (2*)*k*.

End Edit (edit subopcode)

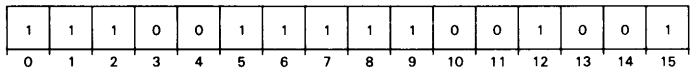
DEND



Terminates the *Edit* subprogram.

Dequeue a Queue Data Element

DEQUE



Dequeues a data element.

AC0 contains the effective address of a queue descriptor.

AC1 specifies the element to be dequeued. The instruction dequeues an element by writing -1 in the forward and backward links. If the contents of AC1 $\neq$ -1, then the accumulator contains the effective address of the data element to be deleted. If the contents of AC1 = -1, then the accumulator deletes the head element of the queue (as obtained from the queue descriptor pointed to by the effective address in AC0).

The instruction first reads all of the links required to complete the dequeuing operation. If a page fault occurs, the instruction restarts at the beginning of the link.

The **DEQUE** instruction requires -- in addition to page zero of the ring of execution for this instruction -- eight pages to be resident, in the worst case, before the instruction will complete. Therefore, nine pages may be required to be resident by this instruction. The worst case occurs when inserting an element between two other elements and when all of the elements and the queue header have one of their affected links on a page boundary.

When all of the required pages are resident, the instruction then attempts to dequeue the data element.

- If dequeuing from an empty queue, then AC1 remains unchanged. If dequeuing the last data element, then the instruction updates AC1 with the address of the data element that was just dequeued. In either case, the next sequential word is executed.
- If dequeuing a data element from a queue containing two or more data elements, then the instruction updates AC1 with the address of the dequeued data element. The next sequential word is skipped.

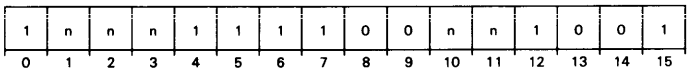
If the data element was the head or tail of the queue, the instruction updates the queue descriptor appropriately.

The instruction checks all reads and writes of links in data elements and queue descriptors against the current ring. Ring numbers of the link addresses must be greater than or equal to the current ring.

The dequeue operation is not interruptible. The entire operation finishes before any interrupts can be enabled. The contents of AC0, AC1, AC2, and AC3 remain unchanged. Carry is unchanged and *overflow* is 0.

Detected Error

DERR *nn*



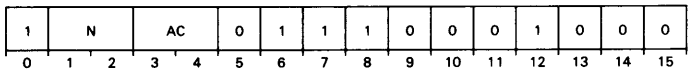
The DERR instruction pushes onto the wide stack the PC of the DERR instruction and a 5-bit code. The instruction then jumps to a user-supplied error (or trap) handler. The processor zero extends the 5-bit error code to 32 bits before pushing it on the stack.

The one word pointer in page zero of the current segment is nonindirectable. When Bit 0 of the pointer is one, the DERR instruction checks for a stack overflow; when Bit 0 is zero, the instruction ignores a stack overflow. Refer to the appropriate functional characteristics manual for additional pointer information.

Carry is unchanged and *overflow* is 0.

Double Hex Shift Left

DHXL *n,ac*



Shifts the 32-bit contents of two accumulators left 1 to 4 hex digits, depending on the value of a 2-bit number in the instruction.

Shifts the 32-bit number contained in bits 16–31 of AC and bits 16–31 of AC+1 left a number of hex digits depending upon the immediate field n. The number of digits shifted is equal to n+1. Bits shifted out are lost and the vacated bit positions are filled with zeroes. Carry remains unchanged and *overflow* is 0.

Bits 0–15 of the modified accumulator are undefined after completion of this instruction.

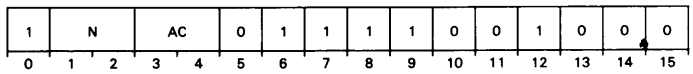
NOTES: *If AC is specified as AC3, then AC+1 is AC0.*

The assembler takes the coded value of n *and subtracts one from it before placing it in the immediate field. Therefore, the programmer should code the exact number of hex digits to be shifted.*

If n is equal to 4, the contents of AC+1 are placed in AC and AC+1 is filled with zeroes.

Double Hex Shift Right

DHXR *n,ac*



Shifts the 32-bit contents of two accumulators right 1 to 4 hex digits, depending on the value of a 2-bit number in the instruction.

Shifts the 32-bit number contained in bits 16–31 of AC and bits 16–31 of AC+1 right a number of hex digits depending upon the immediate field N. The number of digits shifted is equal to N+1. Bits shifted out are lost and the vacated bit positions are filled with zeroes. Carry remains unchanged and *overflow* is 0.

Bits 0–15 of the modified accumulator are undefined after completion of this instruction.

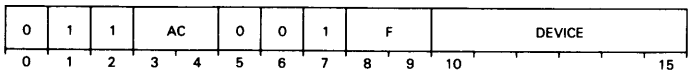
NOTES: *If AC is specified as AC3, then AC+1 is AC0.*

The assembler takes the coded value of n *and subtracts one from it before placing it in the immediate field. Therefore, the programmer should code the exact number of hex digits to be shifted.*

If n is equal to 4, the contents of AC are placed in AC+1 and AC is filled with zeroes.

Data In A

DIA*[f]* *ac,device*



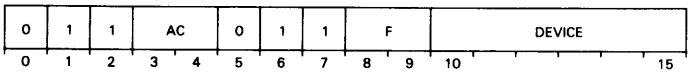
Transfers data from the A buffer of an I/O device to bits 16–31 of an accumulator.

The contents of the A input buffer in the specified device are placed in bits 16–31 of the specified accumulator. After the data transfer, the Busy and Done flags are set according to the function specified by *F*.

The number of data bits moved depends upon the size of the buffer and the mode of operation of the device. Bits in the accumulator that do not receive data are set to 0.

Data In B

DIB*[f]* *ac,device*



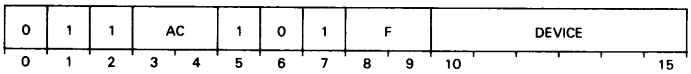
Transfers data from the B buffer of an I/O device to bits 16–31 of an accumulator.

Places the contents of the B input buffer in the specified device in bits 16–31 of the specified accumulator. After the data transfer, sets the Busy and Done flags according to the function specified by *F*.

The number of data bits moved depends upon the size of the buffer and the mode of operation of the device. Bits in the accumulator that do not receive data are set to 0.

Data In C

DIC*[f]* *ac,device*



Transfers data from the C buffer of an I/O device to bits 16–31 of an accumulator.

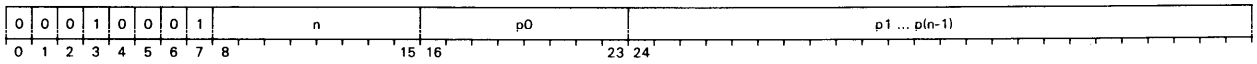
Places the contents of the C input buffer in the specified device in bits 16–31 of the specified accumulator. After the data transfer, sets the Busy and Done flags according to the specified *F*.

The number of data bits moved depends upon the size of the buffer and the mode of operation of the device. Bits in the accumulator that do not receive data are set to 0.

NOTE: *The Assembler and processor reserve the DICC 0,CPU (IORST) instruction for resetting I/O.*

Insert Characters Immediate (edit subopcode)

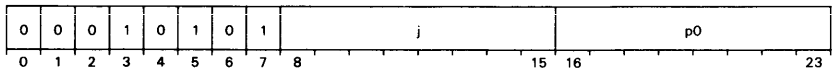
DICI *n,p0[,p1,...,p(n-1)]*



Inserts *n* characters from the opcode stream into the destination field beginning at the position specified by DI. Increases P by $n+2$ and increases DI by *n*.

Insert Character J Times (edit subopcode)

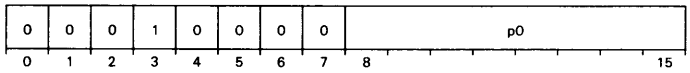
DIMC *j,p0*



Inserts the character specified by *p0* into the destination field a number of times equal to *j* beginning at the position specified by DI. Increases DI by *j*.

Insert Character Once (edit subopcode)

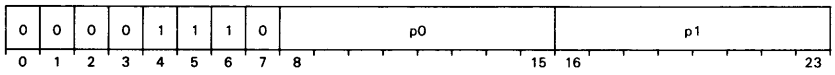
DINC *p0*



Inserts the character specified by *p0* in the destination field at the position specified by DI. Increments DI by one.

Insert Sign (edit subopcode)

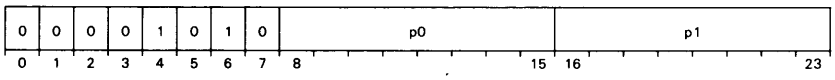
DINS *p0,p1*



If the Sign flag (*S*) is 0, the instruction inserts the character specified by *p0* in the destination field at the position specified by DI. If *S* is 1, the instruction inserts the character specified by *p1* in the destination field at the position specified by DI. Increments DI by one.

Insert Character Suppress (edit subopcode)

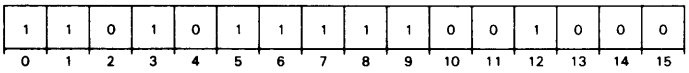
DINT *p0,p1*



If the significance Trigger (*T*) is 0, the instruction inserts the character specified by *p0* in the destination field at the position specified by DI. If *T* is 1, the instruction inserts the character specified by *p1* in the destination field at the position specified by DI. Increments DI by one.

Unsigned Divide

DIV



Divides the unsigned 32-bit integer in bits 16–31 of two accumulators by the unsigned contents of a third accumulator. The quotient and remainder each occupy one accumulator.

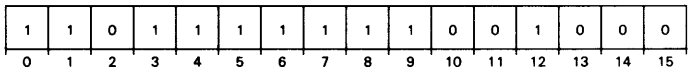
Divides the unsigned, 32-bit number contained in bits 16–31 of AC0 and bits 16–31 of AC1 by the unsigned, 16-bit number in bits 16–31 of AC2. The quotient and remainder are unsigned, 16-bit numbers and are placed in bits 16–31 of AC1 and AC0, respectively. Carry is set to 0. The contents of AC2 remain unchanged. *Overflow* is 0.

Bits 0–15 of the modified accumulator are undefined after completion of this instruction.

NOTE: *Before the divide operation takes place, the number in bits 16–31 of AC0 is compared to the number in bits 16–31 of AC2. If the contents of bits 16–31 of AC0 are greater than or equal to the contents of bits 16–31 of AC2, an overflow condition is indicated. Carry is set to 1 and the operation is terminated. All operands remain unchanged.*

Signed Divide

DIVS



Divides the signed 32-bit integer in bits 16–31 of two accumulators by the signed contents of a third accumulator. The quotient and remainder each occupy one accumulator.

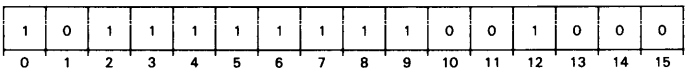
Divides signed, 32-bit two's complement number contained in bits 16–31 of AC0 and bits 16–31 of AC1 by the signed, 16-bit two's complement number in bits 16–31 of AC2. The quotient and remainder are signed, 16-bit numbers and occupy bits 16–31 of AC1 and AC0, respectively. The sign of the quotient is determined by the rules of algebra. The sign of the remainder is always the same as the sign of the dividend, except that a zero quotient or a zero remainder is always positive. Carry is set to 0. The contents of AC2 remain unchanged. *Overflow* is 0.

Bits 0–15 of the modified accumulator are undefined after completion of this instruction.

NOTE: *If the magnitude of the quotient is such that it will not fit into bits 16–31 of AC1, an overflow condition is indicated. Carry is set to 1 and the operation is terminated. The contents of AC0 and AC1 are unpredictable.*

Sign Extend and Divide

DIVX



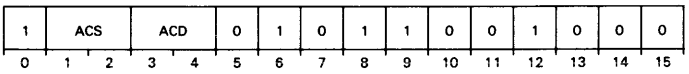
Extends the sign of one accumulator into a second accumulator and performs a *Signed Divide* on the result.

Extends the sign of the 16-bit number in bits 16–31 of AC1 into bits 16–31 of AC0 by placing a copy of bit 16 of AC1 in bits 16–31 of AC0. After extending the sign, the instruction performs a *Signed Divide* operation. *Overflow* is 0.

Bits 0–15 of the modified accumulator are undefined after completion of this instruction.

Double Logical Shift

DLSH *acs,acd*



Shifts the 32-bit contents of two 16-bit accumulators left or right, depending on the contents of a third accumulator.

Shifts the 32-bit number contained in bits 16–31 of ACD and bits 16–31 of ACD+1 either left or right depending on the number contained in bits 24–31 of ACS. The signed, 8-bit two's complement number contained in bits 24–31 of ACS determines the direction of the shift and the number of bits to be shifted. If the number in bits 24–31 of ACS is positive, shifting is to the left; if the number in bits 24–31 of ACS is negative, shifting is to the right. If the number in bits 24–31 of ACS is zero, no shifting is performed. Bits 0–23 of ACS are ignored.

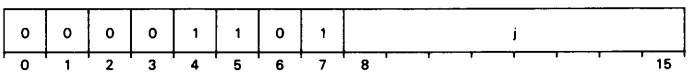
AC3+1 is AC0. The number of bits shifted is equal to the magnitude of the number in bits 24–31 of ACS. Bits shifted out are lost and the vacated bit positions are filled with zeroes. Carry and the contents of ACS remain unchanged. *Overflow* is 0.

Bits 0–15 of the modified accumulator are undefined after completion of this instruction.

NOTE: *If the magnitude of the number in bits 24–31 of ACS is greater than 31_{10}, bits 16–31 of ACD are set to 0. Carry and the contents of ACS remain unchanged.*

Move Alphabetics (edit subopcode)

DMVA *j*

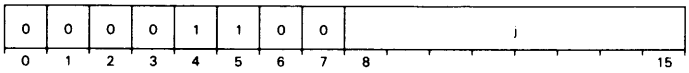


Moves *j* characters from the source field (beginning at the position specified by SI) to the destination field (beginning at the position specified by DI). Increases both SI and DI by *j*. Sets *T* to 1.

Initiates a commercial fault if the attribute specifier word indicates that the source field is data type 5 (packed). Initiates a commercial fault if any of the characters moved is not an alphabetic (A–Z, a–z, or space).

Move Characters (edit subopcode)

DMVC *j*



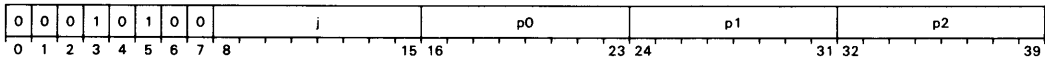
For EDIT, the DMVC instruction increments SI if the source data type is 3 and $j>0$. The instruction then moves *j* characters from the source field beginning at the position specified by SI to the destination field beginning at the position specified by DI. Increases SI and DI by *j*. Sets *T* to 1.

For WEDIT, the DMVC instruction increments SI if the source data type is 3,$j>0$, and SI points to the sign of the source number. The instruction then moves *j* characters from the source field beginning at the position specified by SI to the destination field beginning at the position specified by DI. Increases both SI and DI by *j*. Sets *T* to 1.

Initiates a commercial fault if the attribute specifier word indicates that the source is data type 5 (packed). Performs no validation of the characters.

Move Float (edit subopcode)

DMVF *j,p0,p1,p2*



If the source data type is 3, $j>0$, and SI points to the sign of the source number, the instruction increments SI. Then for *j* characters, the instruction either (depending on T) places a digit substitute in the destination field (beginning at the position specified by DI) or moves a digit from the source field (beginning at the position specified by SI) to the destination field.

When T changes from 0 to 1, the DMVF instruction places in the destination field the digit substitute and the digit. DI increases by j+1. SI increases by the smaller value of either j or the remaining number of digits to move.

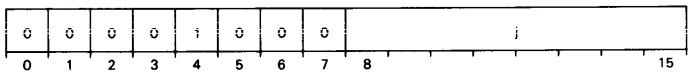
If *T* is 1 or 0, DI increases by j. When *T* is 1, the instruction moves each digit processed from the source field to the destination field. When *T* is 0 and the digit is a zero or space, the instruction places *p0* in the destination field. When *T* is 0 and the digit is a nonzero, the instruction sets *T* to 1 and the characters placed in the destination field depend on *S*. If *S* is 0, the instruction places *p1* in the destination field followed by the digit. If *S* is 1, the instruction places *p2* in the destination field followed by the digit.

If the source data type is 2, the state of SI is undefined after the least significant digit has been processed.

The instruction initiates a commercial fault if any of the digits processed is not valid for the specified data type.

Move Numerics (edit subopcode)

DMVN *j*



The DMVN instruction increments SI by the smaller value of either j or the remaining number of characters to move if the source data type is 3, if J>0, and if SI points to the sign of the source number.

The DMVN instruction moves j characters from the source field (beginning at the position specified by SI) to the destination field (beginning at the position specified by DI.) DI increases by j and T sets to 1. The DMVN instruction moves j characters from the source field beginning at the position specified by SI to the destination field beginning at the position specified by DI.

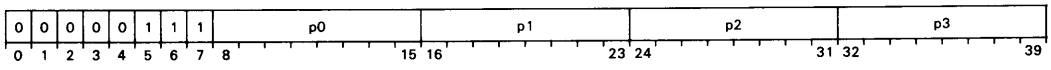
The DMVN instruction increases DI by j and sets T to 1. DMVN compares j to the remaining number of source characters to move and increases SI by the smaller of the two values if the source data type is 3, if j>0, and if SI points to the sign of the source number.

Initiates a commercial fault if any of the characters moved is not valid for the specified data type.

For data type 2, the state of SI is undefined after the least significant digit has been processed.

Move Digit with Overpunch (edit subopcode)

DMVO *p0,p1,p2,p3*



Increments SI if the source data type is 3 and SI points to the sign of the source number. The instruction then either places a digit substitute in the destination field (at the position specified by DI) or moves a digit plus overpunch from the source field (at the position specified by SI) to the destination field (at the position specified by DI). DI and SI always increase by one.

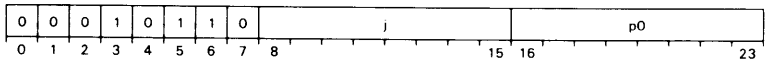
If the source data type is 2, the state of SI is undefined after the least significant digit has been processed.

If the digit is a zero or space and *S* is 0, then the instruction places *p0* in the destination field. If the digit is a zero or space and *S* is 1, then the instruction places *p1* in the destination field. If the digit is a nonzero and *S* is 0, the instruction adds *p2* to the digit and places the result in the destination field. If the digit is a nonzero and *S* is 1, the instruction adds *p3* to the digit and places the result in the destination field. If the digit is a nonzero, the instruction sets *T* to 1. The instruction assumes that *p2* and *p3* are ASCII characters.

The instruction initiates a commercial fault if the character is not valid for the specified data type.

Move Numeric with Zero Suppression (edit subopcode)

DMVS *j,p0*



Increments SI if the source data type is 3, $j>0$, and SI points to the sign of the source number. The instruction then moves *j* characters from the source field (beginning at the position specified by SI) to the destination field (beginning at the position specified by DI). Moves the digit from the source to the destination if *T* is 1. Replaces all zeros and spaces with *p0* as long as *T* is 0. Sets *T* to 1 when the first nonzero digit is encountered. DI always increases by j. SI increases by the smaller value of either j or the remaining number of characters to move.

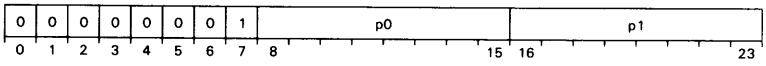
If the source data type is 2, the state of the SI is undefined after the least significant digit has been processed.

This opcode destroys the data type specifier.

Initiates a commercial fault if any of the characters moved is not a numeric (0–9 or space).

End Float (edit subopcode)

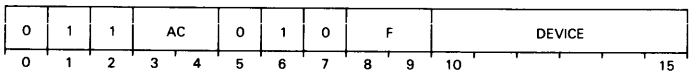
DNDF *p0,p1*



If *T* is 1, the instruction places nothing in the destination field and leaves DI unchanged. If *T* is 0 and *S* is 0, the instruction places *p0* in the destination field at the position specified by DI. If *T* is 0 and *S* is 1, the instruction places *p1* in the destination field at the position specified by DI. It increases DI by 1 and sets *T* to 1.

Data Out A

DOA*[f]* *ac,device*



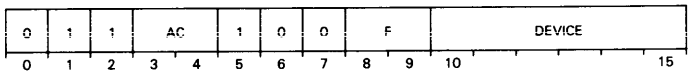
Transfers data from bits 16–31 of an accumulator to the A buffer of an I/O device.

Places the contents of bits 16–31 of the specified accumulator in the A output buffer of the specified device. After the data transfer, sets the BUSY and DONE flags according to the function specified by *F*. The contents of the specified accumulator remain unchanged.

The number of data bits moved depends upon the size of the buffer and the mode of operation of the device.

Data Out B

DOB*[f]* *ac,device*



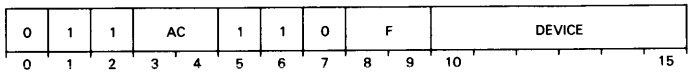
Transfers data from bits 16–31 of an accumulator to the B buffer of an I/O device.

Places the contents of bits 16–31 of the specified accumulator in the B output buffer of the specified device. After the data transfer, sets the BUSY and DONE flags according to the function specified by *F*. The contents of the specified accumulator remain unchanged.

The number of data bits moved depends upon the size of the buffer and the mode of operation of the device.

Data Out C

DOC*[f]* *ac,device*



Transfers data from bits 16–31 of an accumulator to the C buffer of an I/O device.

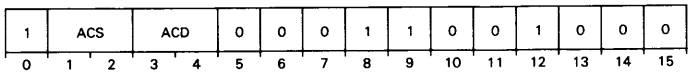
Places the contents of bits 16–31 of the specified accumulator in the C output buffer of the specified device. After the data transfer, sets the BUSY and DONE flags according to the function specified by *F*. The contents of the specified accumulator remain unchanged.

The number of data bits moved depends upon the size of the buffer and the mode of operation of the device.

NOTE: *The Assembler and processor reserve the* **DOC 0,CPU (HALT)** *instruction for stopping the processor.*

Decimal Subtract

DSB *acs,acd*



Performs decimal subtraction on 4-bit binary coded decimal (BCD) numbers and uses carry as a decimal borrow.

Subtracts the unsigned decimal digit contained in ACS bits 28–31 from the unsigned decimal digit contained in ACD bits 28–31. Subtracts the complement of carry from this result. Places the decimal units' position of the final result in ACD bits 28–31 and the complement of the decimal borrow in carry. In other words, if the final result is negative, the instruction indicates a borrow and sets carry to 0. If the final result is positive, the instruction indicates no borrow and sets carry to 1. The contents of ACS and bits 0–27 of ACD remain unchanged. *Overflow* is 0.

Example

Assume that bits 28–31 of AC2 contain 9, bits 28–31 of AC3 contain 7, and carry contains 0. After the instruction **DSB 3,2** is executed, AC3 remains the same; bits 28–31 of AC2 contain 1; and carry is set to 1, indicating no borrow from this *Decimal Subtract* (see Figure 10.3.)

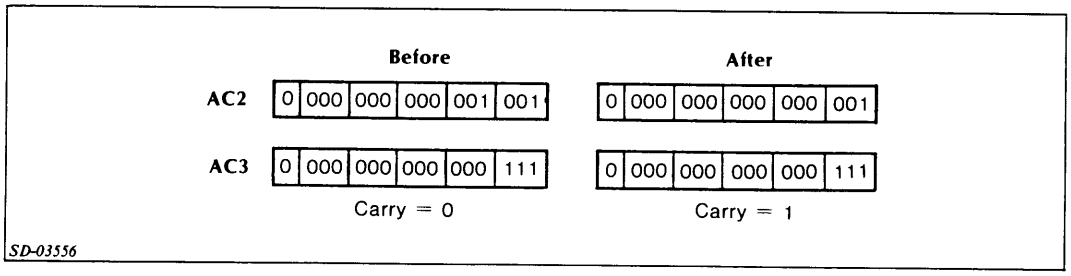
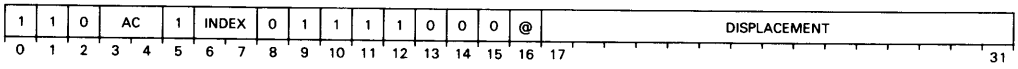


Figure 10.3 DSB example

Dispatch

DSPA *ac,[@]displacement[,index]*



Conditionally transfers control to an address selected from a table.

Computes the effective address, *E* . This is the address of a *dispatch table.* As shown in Figure 10.4, the dispatch table consists of a table of addresses. Immediately before the table are two 16-bit, signed, two's complement limit words, *L* and *H*. The last word of the table is in location *E+H−L*.

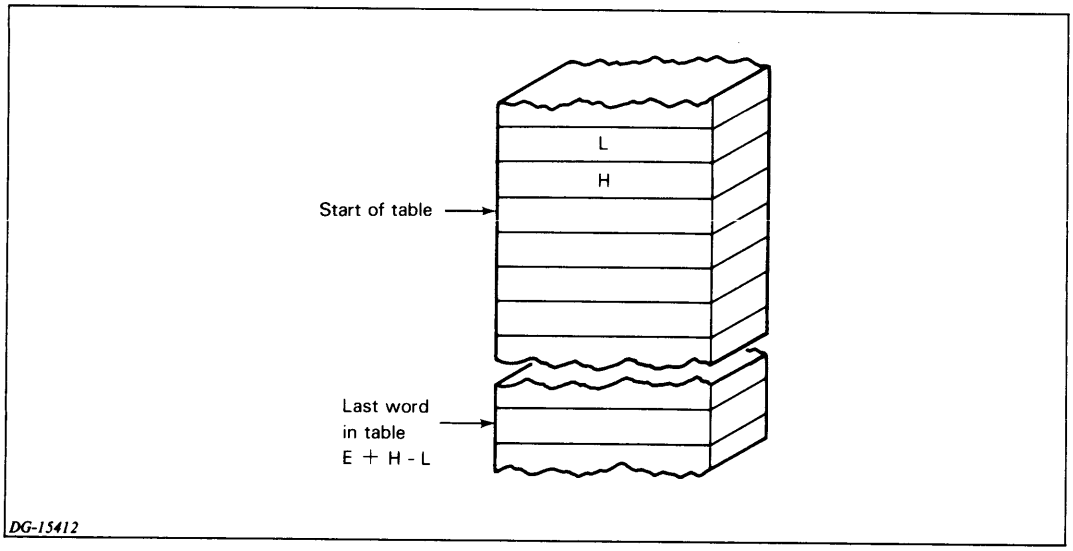


Figure 10.4 DSPA dispatch table structure

Compares the signed, two's complement number contained in bits 16–31 of the specified accumulator to the limit words. If the number in the accumulator is less than *L* or greater than *H*, sequential operation continues with the instruction immediately after the *Dispatch* instruction.

If the number in bits 16–31 of the specified accumulator is greater than or equal to *L* and less than or equal to *H*, the instruction fetches the word at location *E−L+number.*

If the fetched word is equal to 177777_8, sequential operation continues with the instruction immediately after the *Dispatch* instruction. If the fetched word is not equal to 177777_8, the instruction treats this word as the effective address. The instruction places the effective address in the program counter and sequential operation continues with the word addressed by the updated value of the program counter.

The 31-bit effective address generated by this instruction is constrained to be within the first 64 Kbytes of the current segment.

Carry is unchanged and *overflow* is 0.

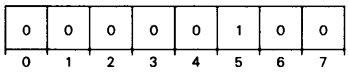
Set S to One (edit subopcode)

DSSO

Sets the Sign flag (*S*) to 1.

Set S to Zero (edit subopcode)

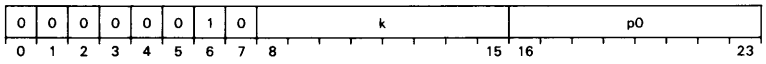
DSSZ



Sets the Sign flag (*S*) to 0.

Store In Stack (edit subopcode)

DSTK *k,p0*

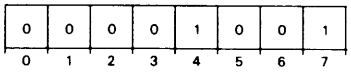


For EDIT, DSTK stores the byte specified by *p0* in bits 8–15 of a word in the narrow stack. Sets bits 0–7 of the word that receives *p0* to 0. If the 8-bit two's complement integer specified by *k* is negative, the instruction addresses the word receiving *p0* by *narrow stack pointer*+1+*k*. If *k* is positive, then the instruction stores *p0* at the address, *narrow frame pointer*+1+*k*.

For WEDIT, DSTK stores the byte specified by *p0* in bits 24-31 of a word in the wide stack. Sets bit 0-23 of the word that receives *p0* to 0. If the 8-bit two's complement integer specified by *k* is negative, the instruction addresses the word receiving *p0* by WSP + 2 + (2*)*k*. If *k* is positive, then the instruction stores *p0* at the address WFP + 2 + (2*)*k*.

Set T to One (edit subopcode)

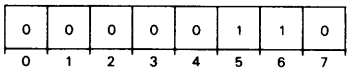
DSTO



Sets the significance Trigger (*T*) to 1.

Set T to Zero (edit subopcode)

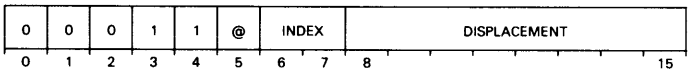
DSTZ



Sets the significance Trigger (*T*) to 0.

Decrement and Skip if Zero

DSZ *[@]displacement[,index]*



Decrements the addressed word, then skips if the decremented value is zero.

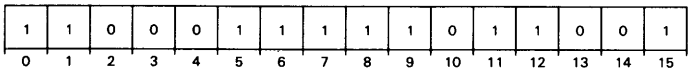
Computes the effective address, *E*. Decrements by one the word addressed by *E* and writes the result back into that location. If the updated value of the location is zero, the instruction skips the next sequential word.

The 31-bit effective address generated by this instruction is constrained to be within the first 64 Kbytes of the current segment.

Overflow is 0 and carry remains unchanged.

Decrement the Word Addressed by WSP and Skip if Zero

DSZTS



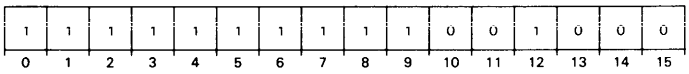
Decrements the double word addressed by the wide stack pointer and skips the next 16-bit word if the decremented value is zero.

Uses the contents of WSP (the wide stack pointer) as the address of a double word. Decrements the contents of the word addressed by WSP. If the decremented value is equal to zero, the instruction skips the next word. Carry is unchanged and *overflow* is 0.

NOTE: *The operation performed by this instruction is not indivisible.*

Load CPU Identification

ECLID

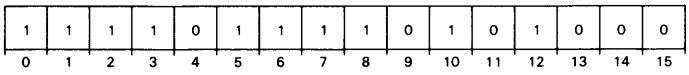


Loads the CPU identification into AC0.

Refer to the specific functional characteristics manual for the accumulator format. Carry is unchanged and *overflow* is 0.

Edit

EDIT



Converts a decimal number from either packed or unpacked form to a string of bytes under the control of an edit subprogram. This subprogram can perform many different operations on the number and its destination field, including leading zero suppression, leading or trailing signs, floating fill characters, punctuation control, and insertion of text into the destination field. The instruction also performs operations on alphanumeric data if data type 4 is specified.

The instruction maintains two flags and three indicators or pointers.

The flags are the significance Trigger (*T*) and the Sign flag (*S*). *T* is set to 1 when the first nonzero digit is processed, unless otherwise specified by an edit opcode. At the beginning of an *Edit* instruction, *T* is set to 0. *S* is set to reflect the sign of the number being processed. If the number is positive, *S* is set to 0. If the number is negative, *S* is set to 1.

The three indicators are the Source Indicator (SI), the Destination Indicator (DI), and the opcode Pointer (P). Each is 16 bits wide and contains a byte pointer to the *current* byte in each respective area. At the beginning of an *Edit* instruction, SI is set to the value contained in bits 16–31 of AC3. DI is set to the value contained in bits 16–31 of AC2 and P is set to the value contained in bits 16–31 of AC0. Also at this time, the sign of the source number is checked for validity.

The subprogram is made up of 8-bit opcodes followed by one or more 8-bit operands. P, a byte pointer, acts as the *program counter* for the *Edit* subprogram. The subprogram proceeds sequentially until a branching operation occurs—much the same way programs are processed. Unless instructed to do otherwise, the *Edit* instruction updates P after each operation to point to the next sequential opcode. The instruction continues to process 8-bit opcodes until directed to stop by the **DEND** opcode.

The subprogram can test and modify *S* and *T* as well as modify SI, DI and P.

Upon entry to **EDIT**, bits 16–31 of AC0 contain a byte pointer to the first opcode of the *Edit* subprogram.

Bits 16–31 of AC1 contain the data-type indicator describing the number to be processed.

Bits 16–31 of AC2 contain a byte pointer to the the first byte of the destination field.

Bits 16–31 of AC3 contain a byte pointer to the first byte of the source field.

The fields may overlap in any way. However, the instruction processes characters one at a time, so unusual side effects may be produced by certain types of overlap.

Upon successful termination, carry contains the significance Trigger; bits 16–31 of AC0 contain a byte pointer (P) to the next opcode to be processed; AC1 is undefined; bits 16–31 of AC2 contain a byte pointer (DI) to the next destination byte; and bits 16–31 of AC3 contain a byte pointer (SI) to the next source byte. *Overflow* is 0.

The 31-bit effective address generated by this instruction is constrained to be within the first 64 Kbytes of the current segment.

NOTES: *If SI is moved outside the area occupied by the source number, zeros will be supplied for numeric moves, even if SI is later moved back inside the source area.*

Some opcodes perform movement of characters from one string to another. For those opcodes which move numeric data, special actions may be performed. For those which move non-numeric data, characters are copied exactly to the destination.

The Edit *instruction places information on the wide stack. Therefore, the wide stack must be set up and have at least nine words available for use.*

If the Edit *instruction is interrupted, it places restart information on the wide stack, sets IRES, sets RES, and places* 177777_8 *in AC0.*

In the description of some of the *Edit* opcodes, we use the symbol j to denote how many characters a certain operation should process. When the high order bit of j is 1, j has a different meaning; it is a pointer into the stack to a word that denotes the number of characters the instruction should process. So, in those cases where the high order bit of j is 1, the instructions interpret j as an 8-bit, two's complement number pointing into the stack. The number on the stack is at the address

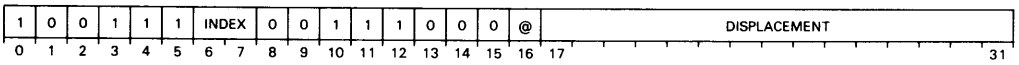
stack pointer + 1 + j.

The operation uses the number at this address as a character count instead of j.

An *Edit* operation that processes numeric data (e.g., **DMVN**) skips a leading or trailing sign code it encounters; similarly, such an operation converts a high-order or low-order sign to its correct numeric equivalent.

Extended Decrement and Skip if Zero

EDSZ *[@]displacement[,index]*



Decrements the addressed word, then skips if the decremented value is zero.

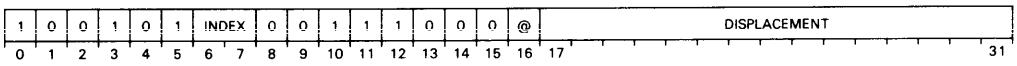
Computes the effective address, E. Decrements by one the contents of the location addressed by E and writes the result back into that location. If the updated value of the word is zero, the instruction skips the next sequential word.

The 31-bit effective address generated by this instruction is constrained to be within the first 64 Kbytes of the current segment.

Carry remains unchanged and *overflow* is 0.

Extended Increment and Skip if Zero

EISZ *[@]displacement[,index]*



Increments the addressed word, then skips if the incremented value is zero.

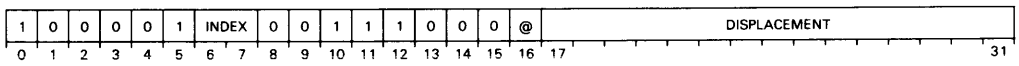
Computes the effective address, *E*. Increments by one the contents of the location specified by *E* and writes the new value back into memory at the same address. If the updated value of the location is zero, the instruction skips the next sequential word.

The 31-bit effective address generated by this instruction is constrained to be within the first 64 Kbytes of the current segment.

Carry remains unchanged and *overflow* is 0.

Extended Jump

EJMP *[@]displacement[,index]*



Loads an effective address into the program counter.

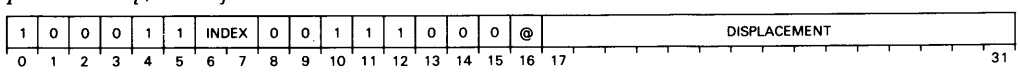
Computes the effective address, *E*, and places it in the program counter. Sequential operation continues with the word addressed by the updated value of the program counter.

The 31-bit effective address generated by this instruction is constrained to be within the first 64 Kbytes of the current segment.

Carry is unchanged and *overflow* is 0.

Extended Jump to Subroutine

EJSR *[@]displacement[,index]*



Increments and stores the value of the program counter in AC3, then places a new address in the program counter.

Computes the effective address, *E*. The instruction then places the address of the next sequential instruction (the instruction following the **EJSR** instruction) in AC3. Places *E* in the program counter. Sequential operation continues with the word addressed by the updated value of the program counter.

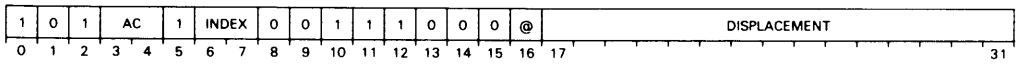
The 31-bit effective address generated by this instruction is constrained to be within the first 64 Kbytes of the current segment.

Overflow is 0 and carry is unchanged.

NOTE: *The instruction computes* E *before it places the incremented program counter in AC3.*

Extended Load Accumulator

ELDA *ac,[@]displacement[,index]*



Copies a word from memory to an accumulator.

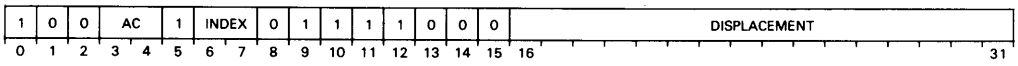
Calculates the effective address, *E*. Places the contents of the location addressed by *E* in bits 16–31 of the specified accumulator. The contents of the location addressed by *E* remain unchanged.

The 31-bit effective address generated by this instruction is constrained to be within the first 64 Kbytes of the current segment.

Carry remains unchanged and *overflow* is 0.

Extended Load Byte

ELDB *ac,displacement[,index]*



Copies a byte from memory into an accumulator.

Forms a byte pointer from the displacement in the following way: shifts the 16-bit number contained in the displacement field to the right one bit, producing a 15-bit address and a 1-bit byte indicator. Uses the value of the index bits to determine an offset value. Adds the offset value to the 15-bit address produced from the displacement to give a memory address. The byte indicator designates which byte of the addressed word will be loaded into bits 24–31 of the specified accumulator. The instruction sets bits 16–23 of the specified accumulator to 0. Carry is unchanged and *overflow* is 0.

The 31-bit effective address generated by this instruction is constrained to be within the first 64 Kbytes of the current segment.

The instruction destroys the previous contents of bits 16–31 of the specified accumulator, but it does not alter either the index value or the displacement.

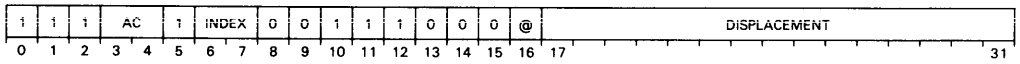
The argument *index* selects the source of the index value. It may have values in the range of 0–3. The following list gives the meaning of each value.

Index Bits	Index Value
0 0	0
0 1	Address of the displacement field (word 2 of this instruction)
1 0	Contents of bits 16-31 of AC2
1 1	Contents of bits 16-31 of AC3

This instruction sets *overflow* to 0 and carry is unchanged.

Extended Load Effective Address

ELEF *ac,[@]displacement[,index]*

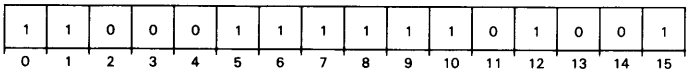


Places an effective address in an accumulator.

Places a 31-bit effective address constrained to be within the first 64 Kbytes of the current segment in an accumulator. Sets bit 0 of the accumulator to 0. *Overflow* is 0 and carry is unchanged.

Enqueue Towards the Head

ENQH



Enqueues a data element.

AC0 contains the effective address of a queue descriptor.

AC1 specifies the element (in the queue) in front of which the new element is added. If the contents of AC1 $\neq$ -1, then the accumulator contains the effective address of the data element in front of which the new element is added. If the contents of AC1 = -1, then the processor adds the new element to the head of the queue (as obtained from the queue descriptor pointed to by the effective address in AC0).

AC2 contains the effective address of the data element to be added to the queue.

The instruction checks the page or pages that contain the current element for valid read and write access privileges. If the privileges are invalid, the appropriate protection fault occurs and the queue remains unchanged.

If the privileges are valid, the instruction checks the queue descriptor addressed by AC0. If the queue descriptor indicates an empty queue, the instruction ignores the contents of AC1, places the data element addressed by AC2 in the queue, and updates the queue descriptor. The next sequential word is executed.

If the descriptor indicates a queue that contains data elements, the instruction first reads all of the links required to complete the enqueue operation. If a page fault occurs, the instruction restarts at the beginning of the link.

The ENQH instruction requires -- in addition to page zero of the ring of execution for this instruction -- eight pages to be resident, in the worst case, before the instruction will complete. Therefore, nine pages may be required to be resident by this instruction. The worst case occurs when inserting an element between two other elements and all of the elements and the queue header have one of their affected links on a page boundary.

When all of the required pages are resident, the instruction then enqueues the data element addressed by AC2 *before* the data element addressed by AC1. If the new data element becomes the head of the queue, the instruction updates the queue descriptor appropriately. The next sequential word is skipped.

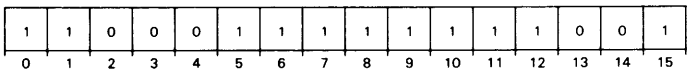
The instruction checks all reads and writes of links in data elements and queue descriptors against the current ring. Ring numbers of the link addresses must be greater than or equal to the current ring.

The enqueue operation is not interruptible. The entire operation completes before any interrupts are enabled.

The instruction leaves the contents of AC0, AC1, AC2, and AC3 unchanged. Carry is unchanged and *overflow* is 0.

Enqueue Towards the Tail

ENQT



Enqueues a data element.

AC0 contains the effective address of a queue descriptor.

AC1 specifies the element (in the queue) in back of which the new element is added. If the contents of AC1 ≠ -1, then the accumulator contains the effective address of the data element in back of which the new element is added. If the contents of AC1 = -1, then the accumulator adds the new element to the tail of the queue (as obtained from the queue descriptor pointed to by the effective address in AC0).

AC2 contains the effective address of the data element to be added to the queue.

The instruction checks the page or pages that contain the current element for valid read and write access privileges. If the privileges are invalid, the appropriate protection fault occurs and the queue remains unchanged.

If the privileges are valid, the instruction checks the queue descriptor addressed by AC0. If the queue descriptor indicates an empty queue, the instruction ignores the contents of AC1 and enqueues the data element addressed by AC2. The instruction updates the queue descriptor if necessary, then the next sequential word is executed.

If the descriptor indicates a queue that contains data elements, the instruction first reads all of the links required to complete the enqueuing operation. If a page fault occurs, the instruction restarts at the beginning of the link.

The ENQT instruction requires -- in addition to page zero of the ring of execution for this instruction -- eight pages to be resident, in the worst case, before the instruction will complete. Therefore, nine pages may be required to be resident by this instruction. The worst case occurs when inserting an element between two other elements and when all of the elements and the queue header have one of their affected links on a page boundary.

When all of the required pages are resident, the instruction then enqueues the data element addressed by AC2 *after* the data element addressed by AC1. If the new data element becomes the tail of the queue, the instruction updates the queue descriptor appropriately. The next sequential word is skipped.

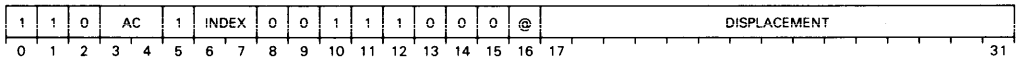
The instruction checks all reads and writes of links in data elements and queue descriptors against the current ring. Ring numbers of the link addresses must be greater than or equal to the current ring.

The enqueue operation is not interruptible. The entire operation completes before any interrupts are enabled.

The instruction leaves the contents of AC0, AC1, AC2, and AC3 unchanged. Carry is unchanged and *overflow* is 0.

Extended Store Accumulator

ESTA *ac,[@]displacement[,index]*



Stores the contents of bits 16–31 of an accumulator into a memory location.

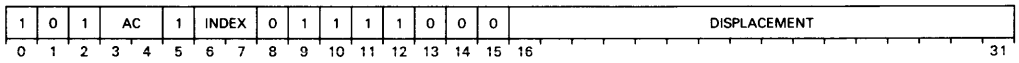
Places contents of bits 16–31 of the specified accumulator in the word addressed by the effective address, *E*. The previous contents of the location addressed by *E* are lost. The contents of the specified accumulator and carry remain unchanged.

The 31-bit effective address generated by this instruction is constrained to be within the first 64 Kbytes of the current segment.

Overflow is 0.

Extended Store Byte

ESTB *ac,displacement[,index]*



Copies into memory the byte contained in bits 24–31 of an accumulator.

Forms a byte pointer from the displacement as follows: shifts the 16-bit number contained in the displacement field to the right one bit, producing a 15-bit address and a 1-bit byte indicator. Uses the value of the index bits to determine an offset value. Adds the offset value to the 15-bit address produced from the displacement field to give a memory address. The byte indicator determines which byte of the addressed location will receive bits 24–31 of the specified accumulator.

The 31-bit effective address generated by this instruction is constrained to be within the first 64 Kbytes of the current segment.

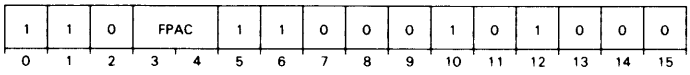
The argument *index* selects the source of the index value. It may have values in the range of 0–3. the following list gives the meaning of each value.

Index Bits	Index Value
0 0	0
0 1	Address of the displacement field (word 2 of this instruction)
1 0	Contents of bits 16-31 of AC2
1 1	Contents of bits 16-31 of AC3

This instruction leaves carry unchanged; *overflow* is 0.

Absolute Value

FAB *fpac*

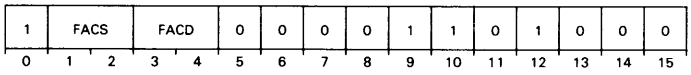


Sets the sign bit of FPAC to 0.

Updates the *Z* and *N* flags in the floating-point status register to reflect the new contents of FPAC.

Add Double (FPAC to FPAC)

FAD *facs,facd*

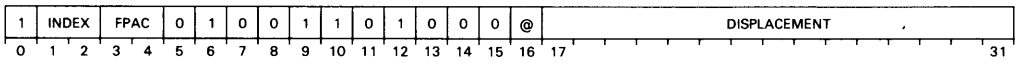


Adds the 64-bit floating-point number in FACS to the 64-bit floating-point number in FACD.

Adds the 64-bit floating-point number in FACS to the 64-bit floating-point number in FACD. Places the normalized result in FACD. Leaves the contents of FACS unchanged and updates the *Z* and *N* flags in the floating-point status register to reflect the new contents of FACD.

Add Double (Memory to FPAC)

FAMD *fpac,[@]displacement[,index]*



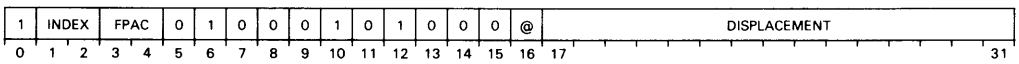
Adds the 64-bit floating-point number in the source location to the 64-bit floating-point number in FPAC and places the normalized result in FPAC.

Computes the effective address, *E*. Uses *E* to address a double-precision (four-word) operand. Adds this 64-bit floating-point number to the 64-bit floating-point number in the specified FPAC. Places the normalized result in the specified FPAC. Leaves the contents of the source location unchanged and updates the *Z* and *N* flags in the floating-point status register to reflect the new contents of FPAC.

The 31-bit effective address generated by this instruction is constrained to be within the first 64 Kbytes of the current segment.

Add Single (Memory to FPAC)

FAMS *fpac,[@]displacement[,index]*



Adds the 32-bit floating-point number in the source location to the 32-bit floating-point number in FPAC and places the normalized result in FPAC.

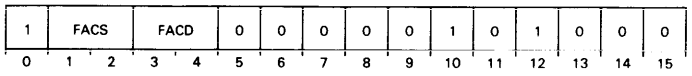
Computes the effective address, *E*. Uses *E* to address a single-precision (double-word) operand. Adds this 32-bit floating-point number to the floating-point number in bits

0–31 of the specified FPAC. Places the normalized result in the specified FPAC. Leaves the contents of the source location unchanged and updates the *Z* and *N* flags in the floating-point status register to reflect the new contents of FPAC. Sets bits 32–63 of FACD to 0.

The 31-bit effective address generated by this instruction is constrained to be within the first 64 Kbytes of the current segment.

Add Single (FPAC to FPAC)

FAS *facs,facd*

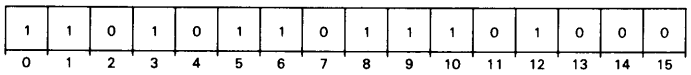


Adds the 32-bit floating-point number in bits 0–31 of FACS to the 32-bit floating-point number in bits 0–31 of FACD.

Adds the 32-bit floating-point number in bits 0–31 of ACS to the 32-bit floating-point number in bits 0–31 of FACD. Places the normalized result in FACD. Leaves the contents of FACS unchanged. Sets bits 32–63 of FACD to 0 and updates the *Z* and *N* flags in the floating-point status register to reflect the new contents of FACD.

Clear Errors

FCLE



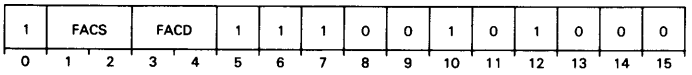
Sets bits 0–4 and bits 28–31 of the floating-point status register to 0.

NOTES: *Since this instruction sets the* ANY *bit of the FPSR to 0, the FPPC field is undefined.*

The **IORST** *instruction and the system reset function will also set these bits to 0.*

Compare Floating Point

FCMP *facs,facd*



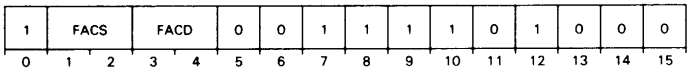
Compares two 64-bit floating-point numbers and sets the *Z* and *N* flags in the floating-point status register accordingly.

Algebraically compares the floating-point numbers in FACS and FACD to each other. Updates the *Z* and *N* flags in the floating-point status register to reflect the result. The contents of FACS and FACD remain unchanged. The following list gives the results of the compare and the corresponding flag settings.

Z	N	Result
1	0	FACS=FACD
0	1	FACS>FACD
0	0	FACS<FACD

Divide Double (FPAC by FPAC)

FDD *facs,facd*

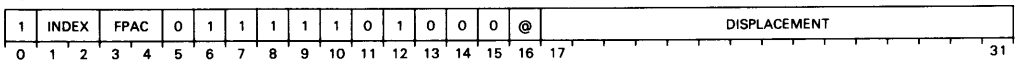


Divides the floating-point number in FACD by the floating-point number in FACS and places the normalized result in FACD.

Divides the 64-bit floating-point number in FACD by the 64-bit floating-point number in FACS. Places the normalized results in FACD. Destroys the previous contents of FACD. Leaves the contents of FACS unchanged and updates the *Z* and *N* flags in the floating-point status register to reflect the new contents of FACD.

Divide Double (FPAC by Memory)

FDMD *fpac,[@]displacement[,index]*



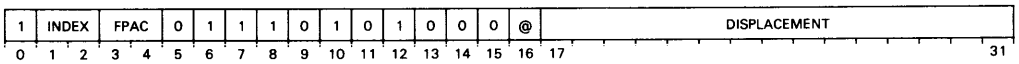
Divides the 64-bit floating-point number in FPAC by the 64-bit floating-point number in the source location and places the normalized result in FPAC.

Computes the effective address, *E*. Uses *E* to address a double-precision (four-word) operand. Divides the 64-bit floating-point number in the specified FPAC by this 64-bit floating-point number. Places the normalized result in the specified FPAC. Leaves the contents of the source location unchanged and updates the *Z* and *N* flags in the floating-point status register to reflect the new contents of FPAC.

The 31-bit effective address generated by this instruction is constrained to be within the first 64 Kbytes of the current segment.

Divide Single (FPAC by Memory)

FDMS *fpac,[displacement[,index]*



Divides the 32-bit floating-point number in bits 0-31 of FPAC by the 32-bit floating-point number in the source location and places the normalized result in FPAC.

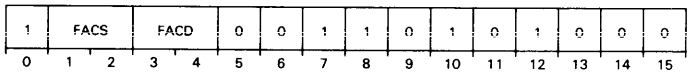
Computes the effective address, *E*. Uses *E* to address a single-precision (double-word) operand. Divides the floating-point number in bits 0-31 of the specified FPAC by this 32-bit floating-point number. Places the normalized result in the specified FPAC. Leaves the contents of the source location unchanged and updates the *Z* and *N* flags in the floating-point status register to reflect the new contents of FPAC.

Sets bits 32–63 of FACD to 0.

The 31-bit effective address generated by this instruction is constrained to be within the first 64 Kbytes of the current segment.

Divide Single (FPAC by FPAC)

FDS *facs,facd*



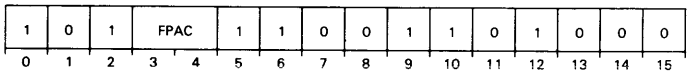
Divides the 32-bit floating-point number in FACD by the 32-bit floating-point number in FACS and places the normalized result in FPAC.

Divides the floating-point number in bits 0–31 of FACD by the floating-point number in bits 0–31 of FACS. Places the normalized result in FACD. Destroys the previous contents of FACD. Leaves the contents of FACS unchanged and updates the *Z* and *N* flags in the floating-point status register to reflect the new contents of FACD.

Sets bits 32–63 of FACD to 0.

Load Exponent

FEXP *fpac*

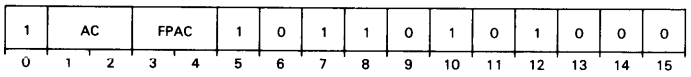


Loads an exponent into bits 1–7 of an FPAC.

Places bits 17–23 of AC0 in bits 1–7 of the specified FPAC. Ignores bits 0–16 and 24–31 of AC0. Changes the *Z* and *N* flags in the floating-point status register to reflect the contents of FPAC. AC0 and bits 0 and 8–63 of FPAC remain unchanged. If FPAC contains true zero, the instruction does not load bits 1–7 of FPAC.

Fix to AC

FFAS *ac,fpac*



Converts the integer portion of the floating-point number contained in the specified FPAC to a signed two's complement integer. Places the result in the specified accumulator.

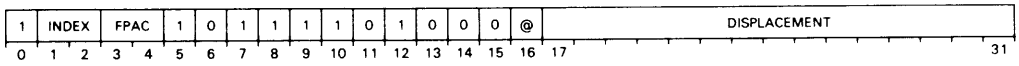
If the integer portion of the number contained in FPAC is less than −32,768 or greater than +32,767, the instruction sets *MOF* in the FPSR to 1. Takes the absolute value of the integer portion of the number contained in the FPAC. Takes the 15 least significant bits of the absolute value and appends a 0 onto the leftmost bit to give a 16-bit number. If the sign of the number is negative, forms the two's complement of the 16-bit result. Places the 16-bit integer in bits 16–31 of the specified accumulator.

If the integer portion is within the range of −32,768 to +32,767 inclusive, the instruction places the 16-bit, two's complement representation of the integer portion of the number contained in the FPAC in bits 16–31 of the specified accumulator.

The instruction leaves the FPAC and the *Z* and *N* flags of the FPSR unchanged.

Fix to Memory

FFMD *fpac,[@]displacement[,index]*



Converts the integer portion of the floating-point number contained in the specified FPAC to a signed two's complement integer. Places the result in a memory location.

Calculates the effective address, *E*. If the integer portion of the number contained in FPAC is less than −2,147,483,648 or greater than +2,147,483,647, the instruction sets *MOF* in the FPSR to 1. Takes the absolute value of the integer portion of the number contained in the FPAC. Takes the 31 least significant bits of the absolute value and appends a 0 onto the leftmost bit to give a 32-bit number. If the sign of the number is negative, forms the two's complement of the 32-bit result. Places the 32-bit integer in the memory locations specified by *E*.

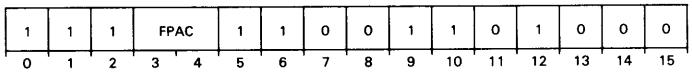
If the integer portion is within the range of −2,147,483,648 to +2,147,483,647 inclusive, the instruction places the 32-bit, two's complement representation of the integer portion of the number contained in the FPAC in the memory locations specified by *E*.

The 31-bit effective address generated by this instruction is constrained to be within the first 64 Kbytes of the current segment.

The instruction leaves the FPAC and the *Z* and *N* flags of the FPSR unchanged.

Halve

FHLV *fpac*



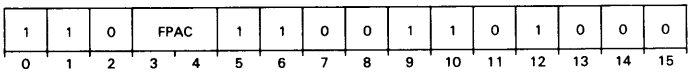
Divides the 64-bit floating-point number in FPAC by 2.

Shifts the mantissa contained in FPAC right one bit position. Fills the vacated bit position with a zero and places the bit shifted out in the guard digit. Normalizes the number and places the result in FPAC. Updates the *Z* and *N* flags in the floating-point status register to reflect the new contents of FPAC.

If underflow occurs, sets the *UNF* flag in the floating-point status register to 1. In this case, the mantissa and sign in FPAC are correct, but the exponent is 128 too large.

Integerize

FINT *fpac*



Sets the fractional part of a floating-point number in the specified 64-bit FPAC to zero and normalizes the result.

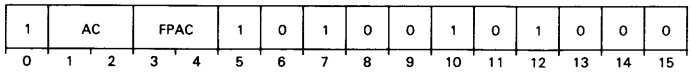
Zeros the fractional portion (if any) of the number contained in the specified FPAC. Normalizes the result. Updates the *Z* and *N* flags in the floating-point status register to reflect the new contents of the specified FPAC.

NOTE: *If the absolute value of the number contained in the specified FPAC is less than 1, the specified FPAC is set to true zero.*

This instruction truncates towards zero and does not do rounding.

Float from AC

FLAS *ac,fpac*

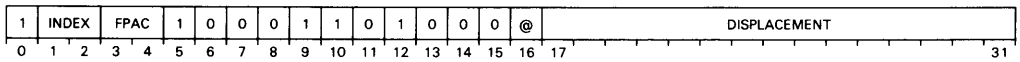


Converts a two's complement number in the range of −32,768 to +32,767 inclusive to floating-point format.

Converts the signed two's complement number contained in bits 16–31 of the specified accumulator to a single-precision floating-point number. Places the result in the high-order 32 bits of the specified FPAC. Sets the low-order 32 bits of the FPAC to 0. Updates the *Z* and *N* flags in the floating-point status register to reflect the new contents of FPAC. The contents of the specified accumulator remain unchanged.

Load Floating-Point Double

FLDD *fpac,[@]displacement[,index]*



Moves four words out of memory and into a specified FPAC.

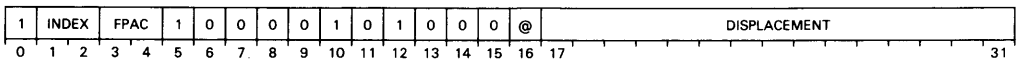
Computes the effective address, *E*. Fetches the double-precision floating-point number at the address specified by *E* and places it in FPAC. Updates the *Z* and *N* flags in the FPSR to reflect the new contents of FPAC.

The 31-bit effective address generated by this instruction is constrained to be within the first 64 Kbytes of the current segment.

NOTE: *This instruction will move unnormalized data without change, but the Z and N flags will be undefined.*

Load Floating-Point Single

FLDS *fpac,[@]displacement[,index]*



Moves two words out of memory into a specified FPAC.

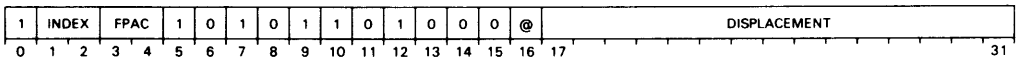
Computes the effective address, *E*. Fetches the single-precision floating-point number at the address specified by *E*. Places the number in the high-order bits of FPAC. Sets the low-order 32 bits of FPAC to 0. Updates the *Z* and *N* flags in the floating-point status register to reflect the new contents of FPAC.

The 31-bit effective address generated by this instruction is constrained to be within the first 64 Kbytes of the current segment.

NOTE: *This instruction will move unnormalized data without change, but the Z and N flags will be undefined.*

Float from Memory

FLMD *fpac,[@]displacement[,index]*



Converts the contents of two 16-bit memory locations to floating-point format and places the result in a specified FPAC.

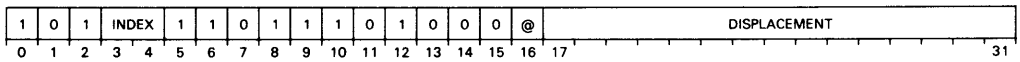
Computes the effective address, *E*. Converts the 32-bit, signed, two's complement number addressed by *E* to a double-precision floating-point number. Places the result in the specified FPAC. Updates the *Z* and *N* flags in the floating-point status register to reflect the new contents of the FPAC.

The 31-bit effective address generated by this instruction is constrained to be within the first 64 Kbytes of the current segment.

The range of numbers that you can convert is −2,147,483,648 to +2,147,483,647 inclusive.

Load Floating-Point Status

FLST *[@]displacement[,index]*



Moves two words out of memory into the floating-point status register.

Computes the effective address, *E*. Places the 32-bit operand addressed by *E* in the floating-point status register as follows:

- Places bits 0–15 of the operand in bits 0–15 of the FPSR. Sets bits 16–32 of the FPSR to 0.
- If *ANY* is 0, bits 33–63 of the FPSR (the FPPC) are undefined.
- If *ANY* is 1, the instruction places the value of the current segment in bits 33–35 of the FPSR, zeroes in bits 36–48, and bits 17–31 of the operand in bits 49–63 of the FPSR.

The 31-bit effective address generated by this instruction is constrained to be within the first 64 Kbytes of the current segment.

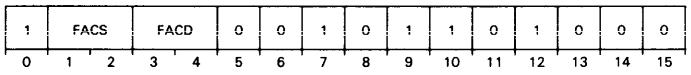
NOTES: *This instruction does not set the* ANY *flag from memory. If any of bits 1–4 are loaded as 1,* ANY *is set to 1; otherwise,* ANY *is 0.*

Bits 12–15 of the FPSR are not set from memory. These bits are the floating-point identification code and cannot be changed.

This instruction initiates a floating-point trap if ANY *and* TE *are both 1 after the FPPC is loaded.*

Multiply Double (FPAC by FPAC)

FMD *facs,facd*

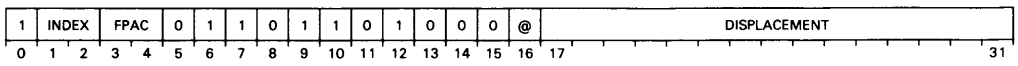


Multiplies the floating-point number in FACD by the floating-point number in FACS and places the normalized result in FACD.

Multiplies the 64-bit floating-point number in FACD by the 64-bit floating-point number in FACS. Places the normalized result in FACD. Leaves the contents of FACS unchanged and updates the *Z* and *N* flags in the floating-point status register to reflect the new contents of FACD.

Multiply Double (FPAC by Memory)

FMMD *fpac,[@]displacement[,index]*



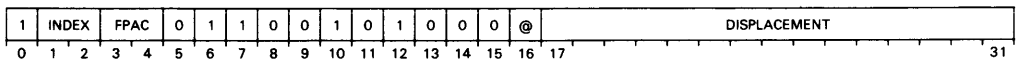
Multiplies the 64-bit floating-point number in the source location by the 64-bit floating-point number in FPAC and places the normalized result in FPAC.

Computes the effective address, *E*. Uses *E* to address a double-precision (four-word) operand. Multiplies this 64-bit floating-point number by the 64-bit floating-point number in the specified FPAC. Places the normalized result in the specified FPAC. Leaves the contents of the source location unchanged and updates the *Z* and *N* flags in the floating-point status register to reflect the new contents of FPAC.

The 31-bit effective address generated by this instruction is constrained to be within the first 64 Kbytes of the current segment.

Multiply Single (FPAC by Memory)

FMMS *fpac,[@]displacement[,index]*



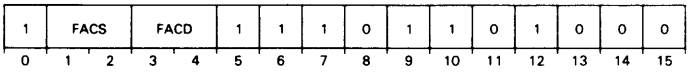
Multiplies the 32-bit floating-point number in the source location by the 32-bit floating-point number in bits 0–31 of FPAC and places the normalized result in FPAC.

Computes the effective address, *E*. Uses *E* to address a single-precision (double-word) operand. Multiplies this 32-bit floating-point number by the floating-point number in bits 0–31 of the specified FPAC. Places the normalized result in bits 0–31 of the specified FPAC. Sets bits 32–63 of FPAC to 0. Leaves the contents of the source location unchanged and updates the *Z* and *N* flags in the floating-point status register to reflect the new contents of FPAC.

The 31-bit effective address generated by this instruction is constrained to be within the first 64 Kbytes of the current segment.

Move Floating Point

FMOV *facs,facd*



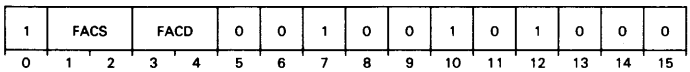
Moves the contents of one 64-bit FPAC to another 64-bit FPAC.

Places the contents of FACS in FACD. Updates the *Z* and *N* flags in the floating-point status register to reflect the new contents of FACD. The contents of FACS remain unchanged.

NOTE: *This instruction will move unnormalized data without change, but the Z and N flags will be undefined.*

Multiply Single (FPAC by FPAC)

FMS *facs,facd*

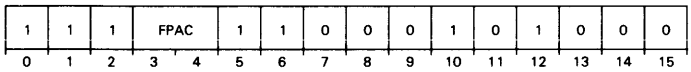


Multiplies the 32-bit floating-point number in bits 0–31 of FACS by the 32-bit floating-point number in bits 0–31 of FACD.

Multiplies the 32-bit floating-point number in bits 0–31 of FACS by the 32-bit floating-point number in bits 0–31 of FACD. Places the normalized result in FACD. Leaves the contents of FACS unchanged. Sets bits 32–63 of FACD to 0 and updates the *Z* and *N* flags in the floating-point status register to reflect the new contents of FACD.

Negate

FNEG *fpac*



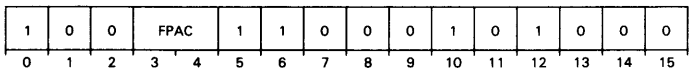
Inverts the sign bit of the FPAC.

Leaves bits 1–63 of FPAC unchanged. Updates the *Z* and *N* flags in the floating-point status register to reflect the new contents of FPAC.

If FPAC contains true zero, leaves the sign bit unchanged.

Normalize

FNOM *fpac*



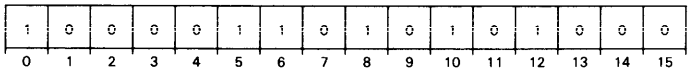
Normalizes the floating-point number in the 64-bit FPAC.

Sets a true zero in FPAC if all the bits of the mantissa are zero. Updates the *Z* and *N* flags in the floating-point status register to reflect the new contents of FPAC.

If an exponent underflow occurs, sets the *UNF* flag in the floating-point status register. In this case, the mantissa and the sign of the number in FPAC are correct, but the exponent is 128 too large.

No Skip

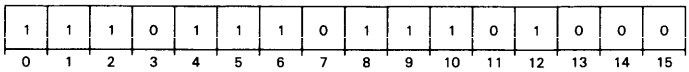
FNS



Executes the next sequential word.

Pop Floating-Point State

FPOP



Pops the state of the floating-point unit off the narrow stack.

Pops an 18-word block off the narrow stack and loads the contents into the FPSR and the four FPACs. The format of the 18-word block is shown in Figure 10.5.

The instruction pops the first 32-bit operand on the stack and places it in the FPSR as follows:

- Places bits 0–15 of the operand in bits 0–15 of the FPSR. Sets bits 16–32 of the FPSR to 0.
- If *ANY* is 0, bits 33–63 of the FPSR (the FPPC) are undefined.
- If *ANY* is 1, the instruction places the value of the current segment in bits 33–35 of the FPSR, zeroes in bits 36–48, and bits 17–31 of the operand in bits 49–63 of the FPSR.

The rest of the stack words are popped in the usual way.

NOTES: *This instruction moves unnormalized data without change.*

This instruction does not set the ANY *flag from memory. If any of bits 1–4 are loaded as 1,* ANY *is set to 1; otherwise,* ANY *is 0.*

Bits 12–15 of the FPSR are not set from memory. These bits are the floating-point identification code and cannot be changed. Refer to the specific functional characteristics manual for the code to use.

This instruction does not initiate a floating-point trap under any conditions of the FPSR.

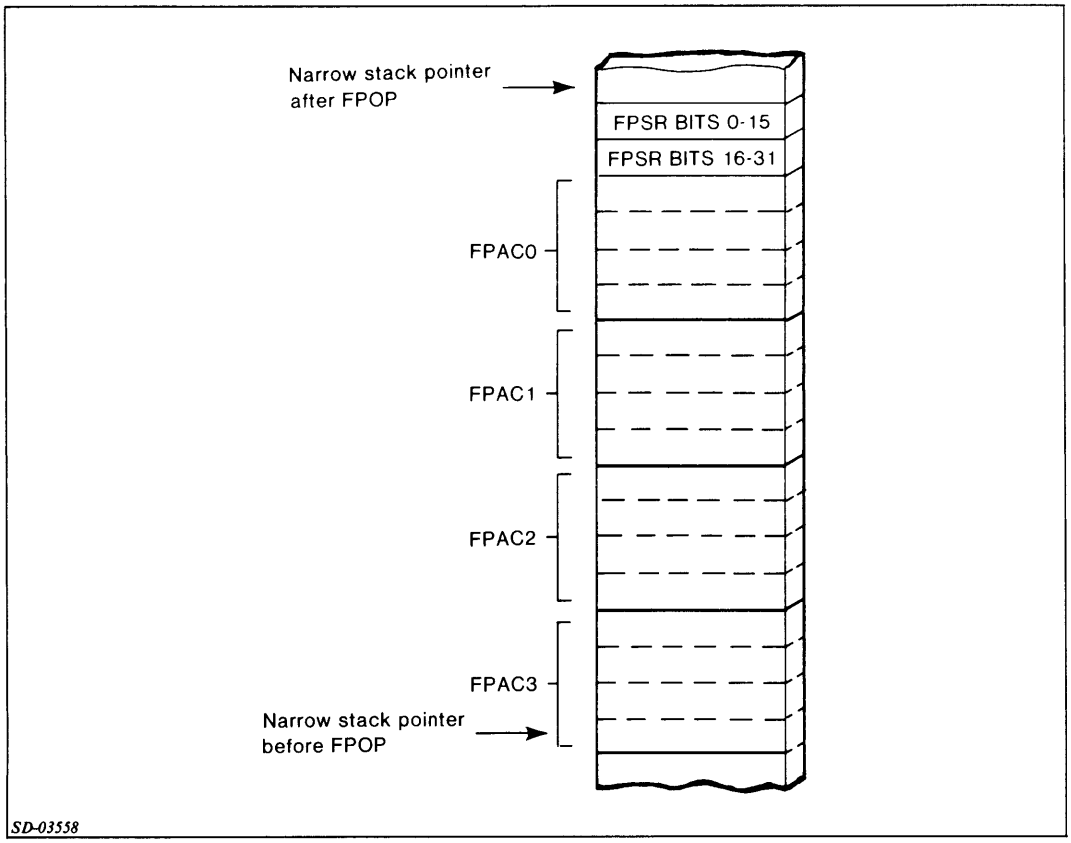


Figure 10.5 Narrow stack, 18-word block

Push Floating-Point State

FPSH

1	1	1	0	0	1	1	0	1	1	1	0	1	0	0	0
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Pushes an 18-word floating-point return block onto the narrow stack, leaving the contents of the floating-point accumulators and the floating-point status register unchanged. The format of the 18 words pushed is illustrated in Figure 10.6.

The instruction pushes the contents of the FPSR as follows:

- Stores bits 0–15 of the FPSR in the first memory word.
- If *ANY* is 0, the contents of the second memory word are undefined.
- If *ANY* is 1, the instruction stores bits 48–63 of the FPSR into the second memory word.

The rest of the block is pushed after the FPSR has been pushed.

NOTES: *This instruction moves unnormalized data without change.*

This instruction does not initiate a floating-point trap under any conditions of the FPSR.

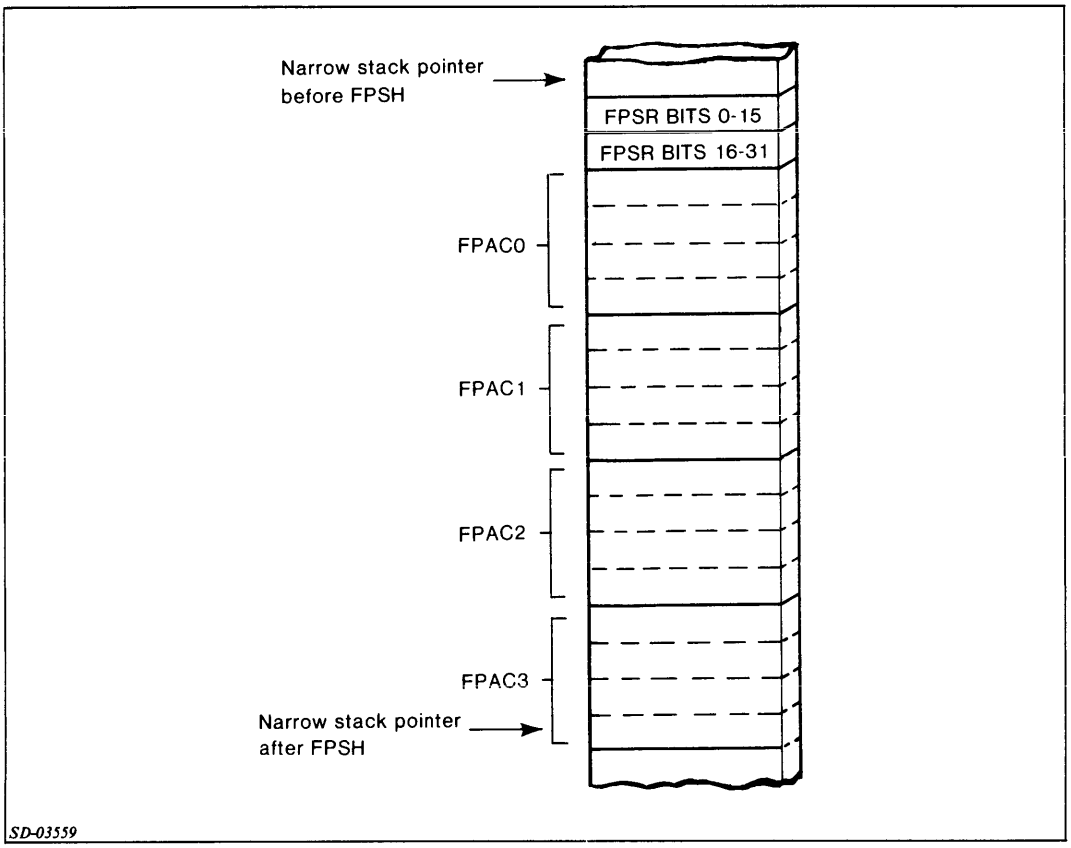
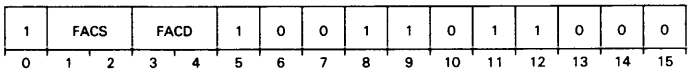


Figure 10.6 Narrow stack, 18-word floating-point return block

Floating-Point Round Double to Single

FRDS *facs,facd*



When FPSR(8) is set to a 1, this instruction rounds a 64-bit floating-point number in FACS to a 32-bit floating-point number and places the result in bits 0–31 of FACD. When FPSR(8) is set to a 0, the instruction moves FACS to FACD and zeroes bits 32–63 of FACD.

Rounds bits 0–31 of FACS using bits 32–63 of FACS. Call FACS<8-31> the "unrounded mantissa" and call FACS<32-63> the "rounding digits." The rounding digits can fall into three ranges:

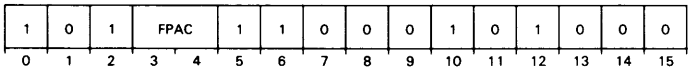
- 0 to $7FFFFFFF_{16}$ inclusive. The result mantissa is equal to the unrounded mantissa without change.
- 80000000_{16}. The result mantissa is formed by adding the least significant bit of the unrounded mantissa to the unrounded mantissa.
- 80000001_{16} to $FFFFFFFF_{16}$ inclusive. The result mantissa is equal to the unrounded mantissa plus 1.

Algorithm is similar to unbiased rounding except that it uses eight rounding digits instead of two guard digits.

Forms the 32-bit result by normalizing the result mantissa and appending the FACS sign and exponent (bits 0–7). Places the 32-bit result in bits 0–31 of FACD. Sets bits 32–63 of FACD to 0 and updates the *Z* and *N* flags in the floating-point status register to reflect the new contents of FACD.

Read High Word

FRH *fpac*

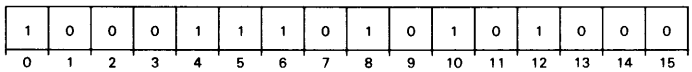


Zero extends the value in the high-order 16 bits of FPAC to 32 bits and places it in AC0. FPAC and the *Z* and *N* flags in the floating-point status register remain unchanged.

NOTE: *This instruction moves unnormalized data without change.*

Skip Always

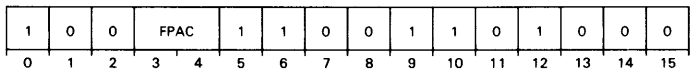
FSA



Skips the next sequential word.

Scale

FSCAL *fpac*



Shifts the mantissa of the 64-bit floating-point number in FPAC either right or left, depending upon the contents of bits 17–23 of AC0. Leaves the contents of AC0 unchanged.

Bits 17–23 of AC0 contain an exponent.

The instruction subtracts the exponent of the number contained in FPAC from the exponent in AC0. The difference between the exponents specifies *D*, a number of hex digits.

If *D* is zero or if FPAC is true zero, the instruction updates the *Z* and *N* flags and stops.

If *D* is positive, the instruction shifts the mantissa of the number contained in FPAC to the right by *D* digits.

If *D* is negative, the instruction shifts the mantissa of the number contained in FPAC to the left by *D* digits. Sets the *MOF* flag in the floating-point status register.

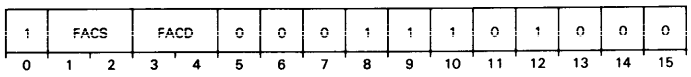
NOTE: *If all the bits in the mantissa are shifted out, FSCAL will set the FPAC to True Zero.*

After the right or left shift, the instruction loads the contents of bits 17–23 of AC0 into the exponent field of FPAC. Bits shifted out of either end of the mantissa are lost. Updates the *Z* and *N* flags in the floating-point status register to reflect the new contents of FPAC.

NOTE: *This instruction does not do rounding.*

Subtract Double (FPAC from FPAC)

FSD *facs,facd*

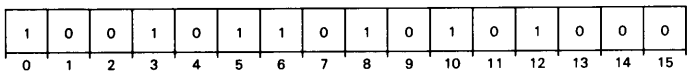


Subtracts the floating-point number in one FPAC from the floating-point number in another FPAC.

Subtracts the 64-bit floating-point number in FACS from the 64-bit floating-point number in FACD. Places the normalized result in FACD. Updates the *Z* and *N* flags in the floating-point status register to reflect the new contents of FACD. The contents of FACS remain unchanged.

Skip on Zero

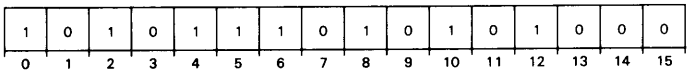
FSEQ



Skips the next sequential word if the *Z* flag of the floating-point status register is 1.

Skip on Greater than or Equal to Zero

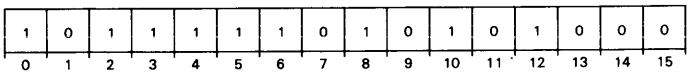
FSGE



Skips the next sequential word if the *N* flag of the floating-point status register is 0.

Skip on Greater than Zero

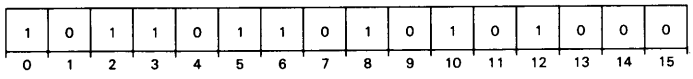
FSGT



Skips the next sequential word if both the *Z* and *N* flags of the floating-point status register are 0.

Skip on Less than or Equal to Zero

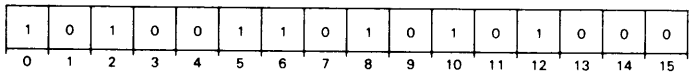
FSLE



Skips the next sequential word if either the *Z* flag or the *N* flag of the floating-point status register is 1.

Skip on Less than Zero

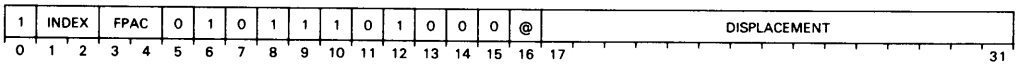
FSLT



Skips the next sequential word if the *N* flag of the floating-point status register is 1.

Subtract Double (Memory from FPAC)

FSMD *fpac,[@]displacement[,index]*



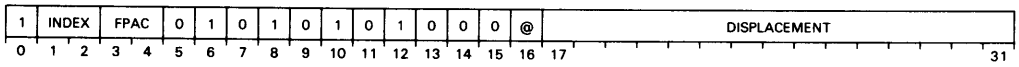
Subtracts the 64-bit floating-point number in the source location from the 64-bit floating-point number in FPAC and places the normalized result in FPAC.

Computes the effective address, *E*. Uses *E* to address a double-precision (four-word) operand. Subtracts this 64-bit floating-point number from the 64-bit floating-point number in the specified FPAC. Places the normalized result in the specified FPAC. Leaves the contents of the source location unchanged and updates the *Z* and *N* flags in the floating-point status register to reflect the new contents of FPAC.

The 31-bit effective address generated by this instruction is constrained to be within the first 64 Kbytes of the current segment.

Subtract Single (Memory from FPAC)

FSMS *fpac,[@]displacement[,index]*



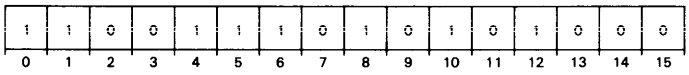
Subtracts the 32-bit floating-point number in the source location from the 32-bit floating-point number in bits 0–31 of FPAC and places the normalized result in FPAC.

Computes the effective address, *E*. Uses *E* to address a single-precision (double word) operand. Subtracts this 32-bit floating-point number from the floating-point number in bits 0–31 of the specified FPAC. Places the normalized result in the specified FPAC. Sets bits 32–63 of FPAC to 0. Leaves the contents of the source location unchanged and updates the *Z* and *N* flags in the floating-point status register to reflect the new contents of FPAC.

The 31-bit effective address generated by this instruction is constrained to be within the first 64 Kbytes of the current segment.

Skip on No Invalid Input Argument

FSND

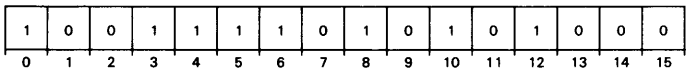


Skips the next sequential word if the *INV* (invalid input argument) flag of the floating-point status register is 0.

NOTE: *This instruction is the previously-designated "Skip on No Zero Divide" instruction and functions identically. When attempting to execute a floating-point divide instruction with a zero divisor, the FPSR INV (formerly DVZ) flag will be set to one and error code 0 will be returned to the FPSR INP bits. The enhanced definition of the INV flag involves the optional Intrinsic Instruction Set (IIS) and is explained in the Faults and Status section of the Floating-Point Computation chapter.*

Skip on Nonzero

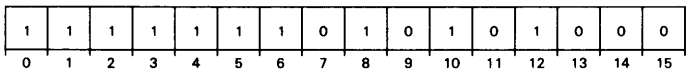
FSNE



Skips the next sequential word if the *Z* flag of the floating-point status register is 0.

Skip on No Error

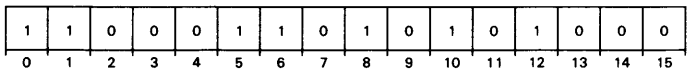
FSNER



Skips the next sequential word if bits 1–4 of the floating-point status register are all 0.

Skip on No Mantissa Overflow

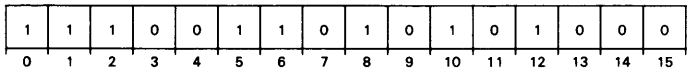
FSNM



Skips the next sequential word if the *MOF* (mantissa overflow) flag of the floating-point status register is 0.

Skip on No Overflow

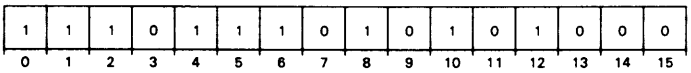
FSNO



Skips the next sequential word if the *OVF* (overflow) flag of the floating-point status register is 0.

Skip on No Overflow and No Invalid Input Argument

FSNOD

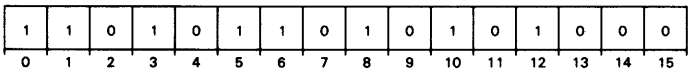


Skips the next sequential word if both the *OVF* flag and the *INV* flag of the floating-point status register are 0.

NOTE: *This instruction is the previously-designated "Skip on No Overflow and No Zero Divide" instruction and functions identically. When attempting to execute a floating-point divide instruction with a zero divisor, the FPSR INV (formerly DVZ) flag will be set to one and error code 0 will be returned to the FPSR INP bits. The enhanced definition of the FPSR INV flag involves the optional Intrinsic Instruction Set (IIS) and is explained in the Faults and Status section of the Floating-Point Computation chapter.*

Skip on No Underflow

FSNU



Skips the next sequential word if the *UNF* (underflow) flag of the floating-point status register is 0.

Skip on No Underflow and No Invalid Input Argument

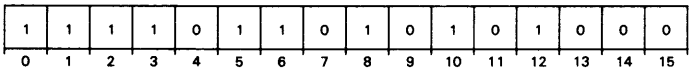
FSNUD

Skips the next sequential word if both the *UNF* flag and the *INV* flag of the floating-point status register are 0.

NOTE: *This instruction is the previously-designated "Skip on No Underflow and No Zero Divide" instruction and functions identically. When attempting to execute a floating-point divide instruction with a zero divisor, the FPSR INV (formerly DVZ) flag will be set to one and error code 0 will be returned to the FPSR INP bits. The enhanced definition of the INV flag involves the optional Intrinsic Instruction Set (IIS) and is explained in the Faults and Status section of the Floating-Point Computation chapter.*

Skip on No Underflow and No Overflow

FSNUO



Skips the next sequential word if both the *UNF* flag and the *OVF* flag of the floating-point status register are 0.

Subtract Single (FPAC from FPAC)

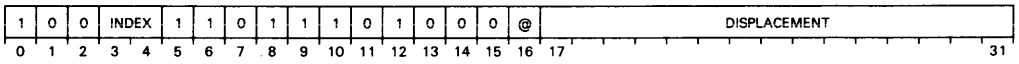
FSS *facs,facd*

Subtracts the floating-point number in one FPAC from the floating-point number in another FPAC.

Subtracts the 32-bit floating-point number in bits 0–31 of FACS from the 32-bit floating-point number in bits 0–31 of FACD. Places the normalized result in bits 0–31 of FACD. Sets bits 32–63 of FACD to 0. Updates the *Z* and *N* flags in the floating-point status register to reflect the new contents of FACD. The contents of FACS remain unchanged.

Store Floating-Point Status

FSST *[@]displacement[,index]*



Moves the contents of the narrow FPSR into memory.

Computes the effective address, *E*, of two sequential, 16-bit locations in memory. Stores the contents of the narrow FPSR in these locations as follows.

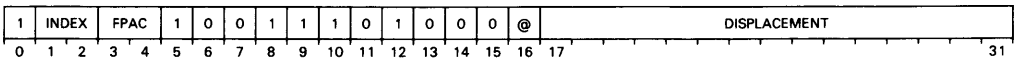
- Stores bits 0–15 of the FPSR in the first memory word.
- If *ANY* is 0, the contents of the second memory word are undefined.
- If *ANY* is 1, the instruction stores bits 48–63 of the FPSR into the second memory word.

The 31-bit effective address generated by this instruction is constrained to be within the first 64 Kbytes of the current segment.

NOTE: *This instruction does not initiate a floating-point trap under any conditions of the FPSR.*

Store Floating-Point Double

FSTD *fpac,[@]displacement[,index]*



Stores the contents of a specified 64-bit FPAC into a memory location.

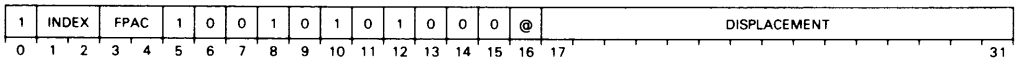
Computes the effective address, *E*. Places the floating-point number contained in FPAC in memory beginning at the location addressed by *E*. Destroys the previous contents of the addressed memory location. The contents of FPAC and the condition codes in the FPSR remain unchanged.

The 31-bit effective address generated by this instruction is constrained to be within the first 64 Kbytes of the current segment.

NOTE: *This instruction will move unnormalized data without change.*

Store Floating-Point Single

FSTS *fpac,[@]displacement[,index]*



Stores the contents of a specified FPAC into a memory location.

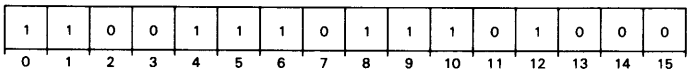
Computes the effective address *E*. Places the 32 high-order bits of FPAC in memory beginning at the location addressed by *E*. Destroys the previous contents of the addressed memory location. The contents of FPAC and the condition codes in the FPSR remain unchanged.

The 31-bit effective address generated by this instruction is constrained to be within the first 64 Kbytes of the current segment.

NOTE: *This instruction will move unnormalized data without change.*

Trap Disable

FTD

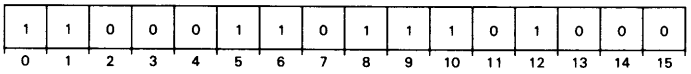


Sets the Trap Enable (*TE*) bit of the FPSR to 0.

NOTE: *The I/O RESET instruction will also set this bit to 0.*

Trap Enable

FTE



Sets the Trap Enable *(TE)* bit of the FPSR to 1.

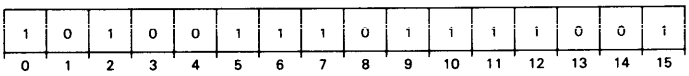
If *ANY* is 1 before execution of this instruction, signals a floating-point trap. If *ANY* is 0 before execution of this instruction, execution continues normally at the end of this instruction.

NOTES: *When this instruction is used to cause a floating-point trap, the FPPC portion of the FPSR will contain the address of the first instruction to cause a fault. Even if another instruction causes a second fault that occurs before the* FTE *instruction executes, the FPPC will still contain the address of the first instruction that caused a fault.*

When a floating-point fault occurs and TE *is 1, the processor sets* TE *to 0 before transferring control to the floating-point error handler.* TE *should be set to 1 before resuming normal processing.*

Fixed-Point Trap Disable

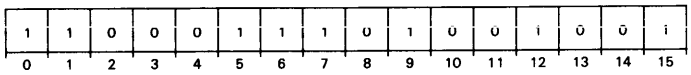
FXTD



Unconditionally sets the *OVK* flag to zero. This disables fixed-point overflow traps. Carry is unchanged.

Fixed-Point Trap Enable

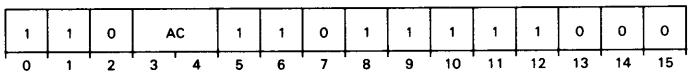
FXTE



Unconditionally sets *OVK* to 1 and *OVR* to 0. This enables fixed-point overflow traps. Carry is unchanged.

Halve

HLV *ac*



Divides the contents of an accumulator by two and rounds the result toward zero.

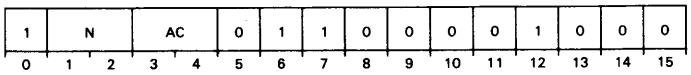
The signed, 16-bit two's complement number contained in bits 16–31 of the specified accumulator is divided by two and rounded toward zero. The result is placed in bits 16–31 of the specified accumulator.

Bits 0–15 of the modified accumulator are undefined after completion of this instruction.

This instruction leaves carry unchanged; *overflow* is 0.

Hex Shift Left

HXL *n,ac*



Shifts the 16-bit contents of an accumulator left 1 to 4 hex digits, depending on the value of a 2-bit number in the instruction.

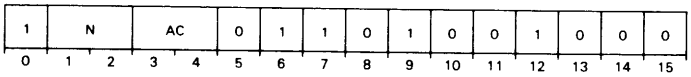
Shifts the contents of bits 16–31 of the specified accumulator left a number of hex digits depending upon the immediate field *N*. The number of digits shifted is equal to $N+1$. Bits shifted out are lost, and the vacated bit positions are filled with zeroes. If *N* is equal to 3, then bits 16–31 of the specified accumulator are shifted out and are set to 0. Leaves carry unchanged. *Overflow* is 0.

Bits 0–15 of the modified accumulator are undefined after completion of this instruction.

NOTE: *The assembler takes the coded value of* n *and subtracts one from it before placing it in the immediate field. Therefore, the programmer should code the exact number of hex digits that he wishes to shift.*

Hex Shift Right

HXR *n,ac*



Shifts the 16-bit contents of an accumulator right 1 to 4 hex digits, depending on the value of a 2-bit number in the instruction.

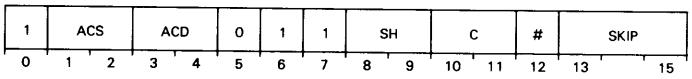
Shifts the contents of bits 16–31 of the specified accumulator right a number of hex digits depending upon the immediate field, *N*. The number of digits shifted is equal to $N+1$. Bits shifted out are lost and the vacated bit positions are filled with zeroes. If *N* is equal to 3, then bits 16–31 of the specified accumulator are shifted out and are set to 0. Leaves carry unchanged. *Overflow* is 0.

Bits 0–15 of the modified accumulator are undefined after completion of this instruction.

NOTE: *The assembler takes the coded value of* n *and subtracts one from it before placing it in the immediate field. Therefore, the programmer should code the exact number of hex digits that he wishes to shift.*

Increment

INC*[c][sh][#] acs,acd[,skip]*



Increments the contents of bits 16–31 of an accumulator.

Initializes carry to the specified value. Increments the unsigned, 16-bit number in bits 16–31 of ACS by one and places the result in the shifter. If the incrementation produces a result that is greater than 32,768, the instruction complements carry. Performs the specified shift operation, and loads the result of the shift into bits 16–31 of ACD if the no-load bit is 0. If the skip condition is true, the next sequential word is skipped.

If the load option is specified, bits 0–15 of ACD are undefined.

NOTE: *If the number in ACS is* 177777_8*, the instruction complements carry.*

For this instruction, *overflow* is 0.

[c]

The processor determines the effect of the CARRY flag (c) on the old value of CARRY before performing the operation (opcode). The following list gives the values of c, bits 10 and 11, and the operation.

Symbol *[c]*	Bits 10-11	Operation
omitted	0 0	Leave CARRY unchanged
Z	0 1	Initialize CARRY to 0
O	1 0	Initialize CARRY to 1
C	1 1	Complement CARRY

[sh]

The processor shifts the CARRY flag and the 16 data bits after performing the instruction operation. The processor can shift the bits left or right one bit position, or it can swap the two bytes. The following list gives the values of sh, bits 8 and 9, and the shift operation.

Symbol *[sh]*	Bits 8-9	Shift Operation
omitted	0 0	Do not shift the result
L	0 1	Shift left
R	1 0	Shift right
S	1 1	Swap the two 8-bit bytes

[#]

Unless you use the no-load option (#), the processor loads the result of the shift operation into the destination accumulator. The no-load option is useful to test the result of the instruction operation without destroying the destination accumulator contents. The following list gives the values of the no-load option, bit 12, and the operation.

Symbol *[#]*	Bit 12	Operation
omitted	0	Load the result into ACD
#	1	Do not load the result and restore the CARRY flag

NOTE: *Do not specify an instruction with the no-load option (#) in combination with either the never skip or always skip option. Thus, the instruction may not end in 1000_2 or 1001_2, other instructions use the bit combinations.*

[skip]

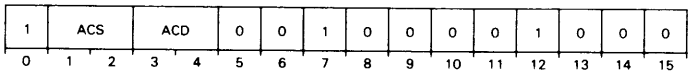
The processor can skip the next instruction if the condition test is true. The following list gives the test conditions, bits 13 to 15, and the operation.

Symbol *[skip]*	Bits 13-15	Operation
omitted	0 0 0	Never skip
SKP	0 0 1	Always skip
SZC	0 1 0	Skip if CARRY is 0
SNC	0 1 1	Skip if CARRY is not 0
SZR	1 0 0	Skip if the result is 0
SNR	1 0 1	Skip if the result is not 0
SEZ	1 1 0	Skip if either CARRY or the result is 0
SBN	1 1 1	Skip if both CARRY and the result are not 0

When the instruction performs a skip, it skips the next sequential 16-bit word. Make sure that a skip does not transfer control to the middle of a 32-bit or longer instruction.

Inclusive OR

IOR *acs,acd*



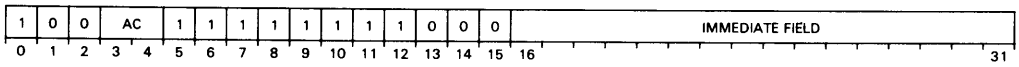
Inclusively ORs the contents of two accumulators.

Forms the logical inclusive OR of the contents of bits 16–31 of ACS and the contents of bits 16–31 of ACD, and places the result in bits 16–31 of ACD. Sets a bit position in the result to 1 if the corresponding bit position in one or both operands contains a 1; otherwise, the instruction sets the result bit to 0. The contents of ACS remain unchanged. Carry remains unchanged. *Overflow* is 0.

Bits 0–15 of the modified accumulator are undefined after completion of this instruction.

Inclusive OR Immediate

IORI *i,ac*



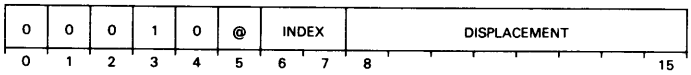
Inclusively ORs the contents of an accumulator with the contents of a 16-bit number in the instructions.

Forms the logical inclusive OR of the contents of the immediate field and the contents of bits 16–31 of the specified accumulator, and places the result in bits 16–31 of the specified accumulator. Carry remains unchanged and *overflow* is 0.

Bits 0–15 of the modified accumulator are undefined after completion of this instruction.

Increment and Skip if Zero

ISZ *[@]displacement[,index]*



Increments the addressed word, then skips if the incremented value is zero.

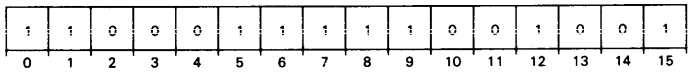
Computes the effective address, *E*. Increments by one the word addressed by *E* and writes the result back into memory at that location. If the updated value of the location is zero, the instruction skips the next sequential word.

The 31-bit effective address generated by this instruction is constrained to be within the first 64 Kbytes of the current segment.

Carry remains unchanged and *overflow* is 0.

Increment the Word Addressed by WSP and Skip if Zero

ISZTS



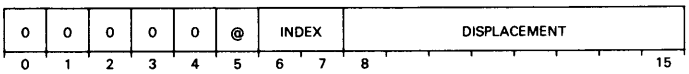
Increments the double word addressed by the wide stack pointer and skips the next 16-bit word if the incremented value is zero.

Uses the contents of WSP (the wide stack pointer) as the address of a double word. Increments the contents of the word addressed by WSP. If the incremented value is equal to zero, then the next sequential word is skipped. Carry is unchanged and *overflow* is 0.

NOTE: *The operation performed by this instruction is not indivisible.*

Jump

JMP *[@]displacement[,index]*



Loads an effective address into the program counter.

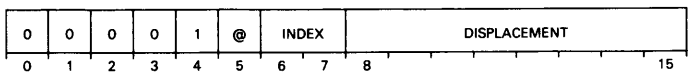
Computes the effective address, *E*, and places it in the program counter. Sequential operation continues with the word addressed by the updated value of the program counter.

The 31-bit effective address generated by this instruction is constrained to be within the first 64 Kbytes of the current segment.

Carry remains unchanged and *overflow* is 0.

Jump to Subroutine

JSR *[@]displacement[,index]*



Increments and stores the value of the program counter in AC3, and then places a new address in the program counter.

Computes the effective address, *E*; then places the address of the next sequential instruction in bits 16–31 of AC3. Places *E* in the program counter. Sequential operation continues with the word addressed by the updated value of the program counter.

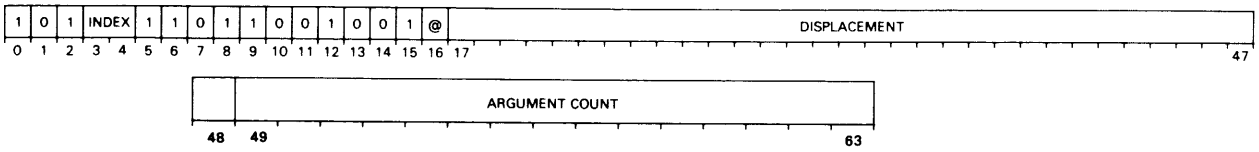
The 31-bit effective address generated by this instruction is constrained to be within the first 64 Kbytes of the current segment.

Carry remains unchanged and *overflow* is 0.

NOTE: *The instruction computes* E *before it places the incremented program counter in AC3.*

Call Subroutine (Long Displacement)

LCALL *[@]displacement[,index[,argument count]]*



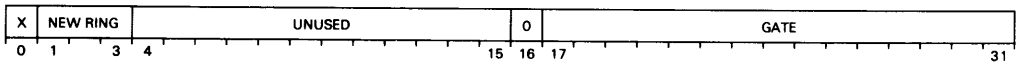
Transfers program control to a subroutine in the current segment or through a gate array in a lower-numbered segment to a subroutine in that lower-numbered segment.

The instruction loads the contents of the PC, plus four, into AC3. The contents of AC3 always reference the current ring.

The effective address (target address) may specify the current ring, an inner ring, or an outer ring.

If the target address specifies an outward ring crossing, a protection fault (code = 7 in AC1) occurs. Note that the contents of the PC in the return block are undefined.

If the target address specifies an inward ring call, then the instruction assumes the target address has the following format:



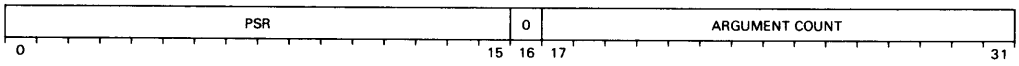
NOTE: *If bit 16 of the inward ring call's target address is not 0 then the results are undefined.*

The instruction checks the gate field of the above format for a legal gate. If the specified gate is illegal, a protection fault (code = 6 in AC1) occurs and no subroutine call is made. Note that the value of the PC in the return block is undefined.

If the specified gate is legal, or if the target address is within the current ring, the instruction then checks the argument count field.

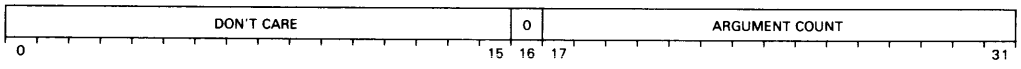
The argument count field specifies the number of arguments the caller has pushed onto the stack. If this field is negative, the caller has also pushed the argument count onto the stack.

If bit 0 of the argument count is 0, the instruction creates a double word with the following format:



The instruction pushes this double word (PSR/argument count) onto the wide stack. If a stack overflow occurs after this push, a stack fault occurs, the PSR is cleared, and no subroutine call is made. Note that the processor uses the inner ring's stack fault handler and the value of the PC in the return block is undefined.

If bit 0 of the argument count is 1 (negative), then the instruction assumes the top double word of the wide stack has the following format:



The instruction uses the argument count on the stack and ignores the argument count coded with the LCALL instruction. The instruction then modifies this double word (PSR/argument count) to include the correct settings of the PSR.

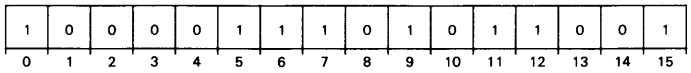
NOTE: *If the target address is in an inner segment, the LCALL instruction copies the number of double words specified in the argument count from the outer segment stack to the inner segment stack, and then pushes the PSR/argument count double word onto the inner stack.*

Regardless of the setting of bit 0 of the argument count, the instruction next unconditionally sets OVR to 0 and loads the PC with the target address. Control then transfers to the word referenced by the PC.

For more information on the LCALL instruction, refer to "Transferring Program Control to Another Segment" in the Program Flow Management chapter.

Load CPU Identification

LCPID

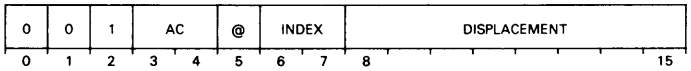


Loads a double word into AC0. Carry is unchanged and *overflow* is 0.

The double word contains the microcode revision level and the main memory size. Refer to the specific functional characteristics manual for the accumulator format.

Load Accumulator

LDA *ac,[@]displacement[,index]*



Copies a word from memory to an accumulator.

Calculates the effective address, *E*. Places the word addressed by *E* in bits 16–31 of the specified accumulator. The contents of the location addressed by *E* remain unchanged.

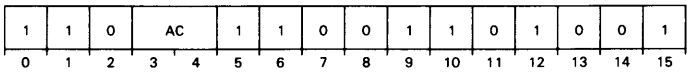
Bits 0–15 are undefined.

The 31-bit effective address generated by this instruction is constrained to be within the first 64 Kbytes of the current segment.

Carry remains unchanged and *overflow* is 0.

Load Accumulator with WFP

LDAFP *ac*

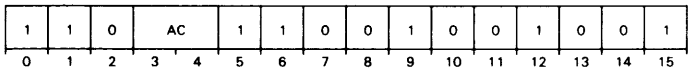


Loads the specified accumulator with the contents of WFP.

Loads the 32-bit contents of WFP (the wide frame pointer) into the specified 32-bit accumulator. Carry is unchanged and *overflow* is 0.

Load Accumulator with WSB

LDASB *ac*

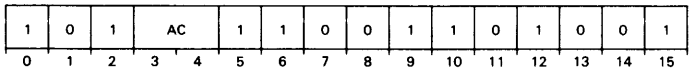


Loads the specified accumulator with the contents of WSB.

Loads the 32-bit contents of WSB (the wide stack base) into the specified 32-bit accumulator. Carry is unchanged and *overflow* is 0.

Load Accumulator with WSL

LDASL *ac*

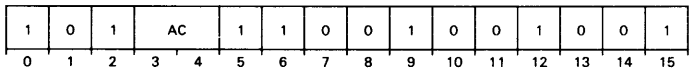


Loads the specified accumulator with the contents of WSL.

Loads the 32-bit contents of WSL (the wide stack limit) into the specified 32-bit accumulator. Carry is unchanged and *overflow* is 0.

Load Accumulator with WSP

LDASP *ac*

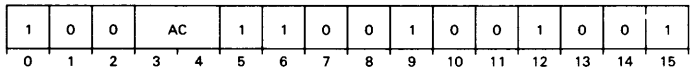


Loads the specified accumulator with the contents of WSP.

Loads the contents of WSP (the wide stack pointer) into the specified accumulator. Carry is unchanged and *overflow* is 0.

Load Accumulator with Double Word

LDATS *ac*

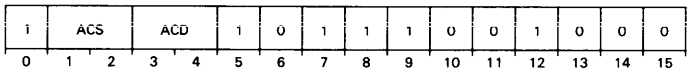


Loads the contents of the word addressed by WSP into an accumulator.

Uses the contents of WSP (the wide stack pointer) as the address of a double word. Loads the contents of the addressed double word into the specified accumulator. Carry is unchanged and *overflow* is 0.

Load Byte

LDB *acs,acd*



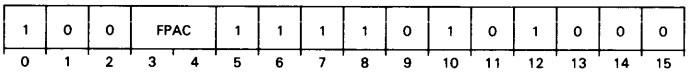
Moves a copy of the contents of a memory byte (as addressed by a byte pointer in one accumulator) into the second accumulator.

Places the 8-bit byte addressed by the byte pointer contained in bits 15–31 of ACS into bits 24–31 of ACD. Sets bits 16–23 of ACD to 0. The contents of ACS remain unchanged unless ACS and ACD are the same accumulator. Carry remains unchanged and *overflow* is 0.

The 31-bit effective address generated by this instruction is constrained to be within the first 64 Kbytes of the current segment.

Load Integer

LDI *fpac*



Translates up to a 16-digit decimal integer from memory to (normalized) floating-point format and places the result in a floating-point accumulator.

Under the control of accumulators AC1 and AC3, converts a decimal integer to floating-point form, normalizes it, and places it in the specified FPAC. The instruction updates the *Z* and *N* bits in the FPSR to describe the new contents of the specified FPAC. Leaves the decimal number unchanged in memory, and destroys the previous contents of the specified FPAC.

Bits 16–31 of AC1 must contain the data-type indicator describing the number.

Bits 16–31 of AC3 must contain a byte pointer which is the address of the high-order byte of the number in memory.

Numbers of data type 7 are not normalized after loading. By convention, the first byte of a number stored according to data type 7 must contain the sign and exponent of the floating-point number. The exponent must be in "excess 64" representation. The instruction copies each byte (following the lead byte) directly to mantissa of the specified FPAC. It then sets to zero each low-order byte in the FPAC that does not receive data from memory.

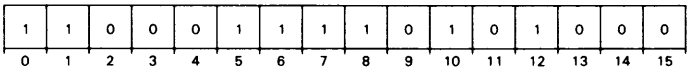
Upon successful completion, the instruction leaves accumulators AC0 and AC1 unchanged. AC2 contains the original contents of AC3. AC3 points to the first byte following the integer field. Carry remains unchanged and *overflow* is 0.

The 31-bit effective address generated by this instruction is constrained to be within the first 64 Kbytes of the current segment.

NOTE: *An attempt to load a minus 0 sets the specified FPAC to true zero.*

Load Integer Extended

LDIX



Distributes a decimal integer of data type 0, 1, 2, 3, 4, or 5 into the four FPACs.

Extends the integer with high-order zeros until it is 32 digits long. Divides the integer into four units of 8 digits each and converts each unit to a floating-point number. Places the number obtained from the 8 high-order digits into FAC0, the number obtained from the next 8 digits into FAC1, the number obtained from the next 8 digits into FAC2, and the number obtained from the low-order 8 bits into FAC3. The instruction places the sign of the integer in each FPAC unless that FPAC has received 8 digits of zeros, in which case the instruction sets FPAC to true zero. The *Z* and *N* flags in the floating-point status register are unpredictable.

Bits 16–31 of AC1 must contain the data-type indicator describing the integer.

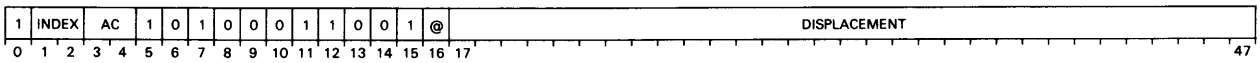
Bits 16–31 of AC3 must contain a byte pointer which is the address of the high-order byte of the integer.

Upon successful termination, the contents of AC0 and AC1 remain unchanged; and AC2 contains the original contents of AC3. AC3 points to the first byte following the integer field. Carry remains unchanged and *overflow* is 0.

The 31-bit effective address generated by this instruction is constrained to be within the first 64 Kbytes of the current segment.

Dispatch (Long Displacement)

LDSP *ac,[@]displacement[,index]*



Dispatches through a table of 28-bit address offsets indexed by the 31-bit PC.

Computes the effective address *E* . This is the address of a *dispatch table.* The dispatch table consists of a table of 28-bit self-relative addresses (bits 0–3 are ignored). Immediately before the table are two signed, two's complement limit double words, *L* and *H*. The last double word of the table is in location $E + 2(H - L)$. The instruction adds the 28-bit self-relative offset in the table entry to the address of the table entry. The segment field of the fetched table entry is ignored.

Compares the signed, two's complement number contained in the accumulator to the signed limit double word. If the number in the accumulator is less than *L* or greater than *H*, sequential operation continues with the instruction immediately after the *Dispatch* instruction.

If the number in AC is greater than or equal to *L* and less than or equal to *H*, the instruction fetches the double word at location $E + 2(number - L)$. If the fetched double word is equal to 37777777777_8 (all 1's), sequential operation continues with the instruction immediately after the *Dispatch* instruction. If the fetched double word is not equal to 37777777777_8, the instruction adds this double word to $E + 2(number - L)$

and places the new address in the program counter. Sequential operation continues with the word addressed by the updated value of the program counter. Carry is unchanged and *overflow* is 0.

Wraparound occurs within the 28-bit offset. A segment crossing cannot occur. The effective address, *E*, references a table of self-relative offsets in the current segment. Thus, bits 1–3 of *E* are always interpreted as the current segment.

The structure of the dispatch table is shown in Figure 10.7.

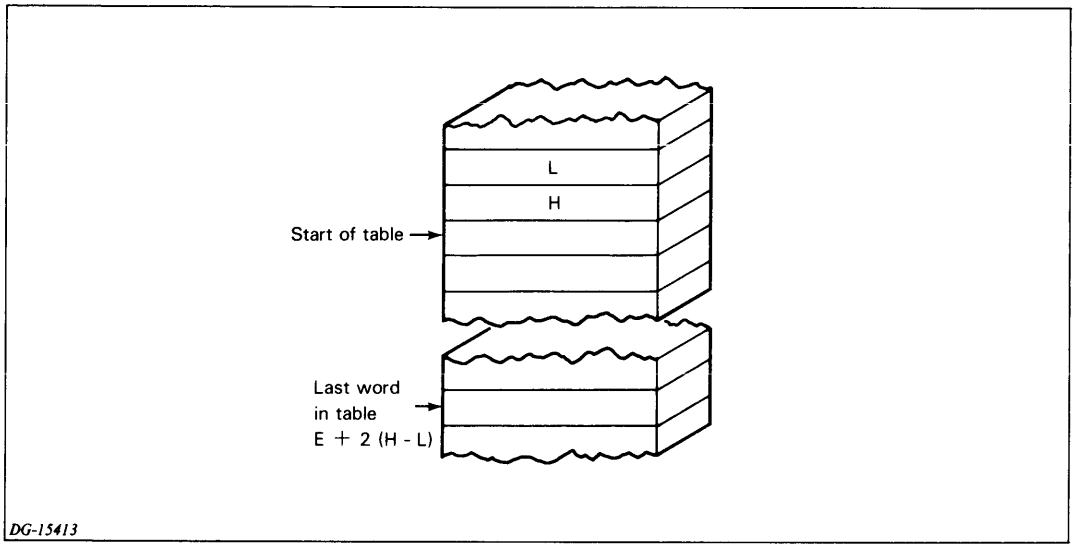
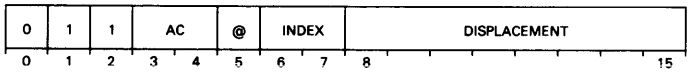


Figure 10.7 LDSP dispatch table structure

Load Effective Address

LEF *ac,[@]displacement[,index]*



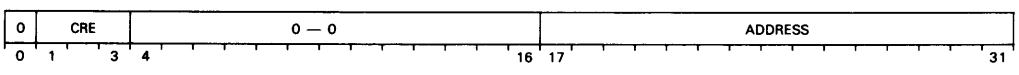
Places an effective address in an accumulator.

Computes the effective address, *E*, within the current segment and places it in the specified accumulator. Sets bit 0 of the accumulator to 0. The previous contents of the AC are lost.

The 31-bit effective address generated by this instruction is constrained to be within the first 64 Kbytes of the current segment (CRE).

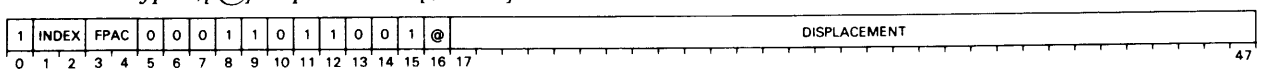
NOTE: *The LEF instruction can only be executed when the address translator is enabled and when the LEF mode is enabled in the segment base register. Otherwise, the processor checks the I/O validity flag when the address translator is enabled. If I/O is enabled or the address translator is disabled, the processor executes the instruction as an I/O instruction. Otherwise, a protection violation occurs.*

Upon instruction completion, carry is unchanged, *overflow* is 0, and the specified accumulator contains the following:



Add Double (Memory to FPAC) (Long Displacement)

LFAMD *fpac,[@]displacement[,index]*

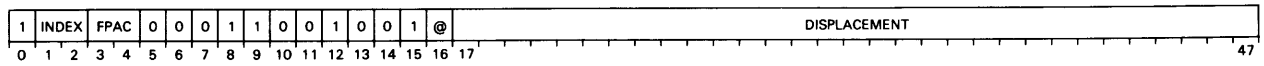


Adds the 64-bit floating-point number in the source location to the 64-bit floating-point number in FPAC and places the normalized result in FPAC.

Computes the effective address, *E*. Uses *E* to address a double-precision (four-word) operand. Adds this 64-bit floating-point number to the 64-bit floating-point number in the specified FPAC. Places the normalized result in the specified FPAC. Leaves the contents of the source location unchanged and updates the *Z* and *N* flags in the floating-point status register to reflect the new contents of FPAC.

Add Single (Memory to FPAC) (Long Displacement)

LFAMS *fpac,[@]displacement[,index]*



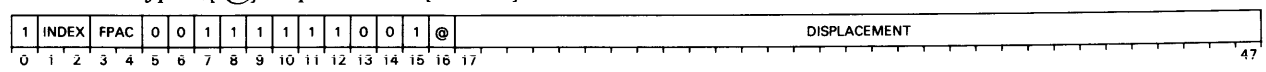
Adds the 32-bit floating-point number in the source location to the 32-bit floating-point number in FPAC and places the normalized result in FPAC.

Computes the effective address, *E*. Uses *E* to address a single-precision (double-word) operand. Adds this 32-bit floating-point number to the floating-point number in bits 0–31 of the specified FPAC. Places the normalized result in the specified FPAC. Leaves the contents of the source location unchanged and updates the *Z* and *N* flags in the floating-point status register to reflect the new contents of FPAC.

Sets bits 32–63 of FACD to 0.

Divide Double (FPAC by Memory) (Long Displacement)

LFDMD *fpac,[@]displacement[,index]*

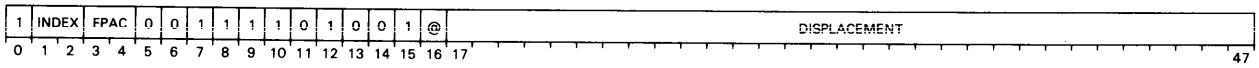


Divides the 64-bit floating-point number in FPAC by the 64-bit floating-point number in the source location and places the normalized result in FPAC.

Computes the effective address, *E*. Uses *E* to address a double-precision (four-word) operand. Divides the 64-bit floating-point number in the specified FPAC by this 64-bit floating-point number. Places the normalized result in the specified FPAC. Leaves the contents of the source location unchanged and updates the *Z* and *N* flags in the floating-point status register to reflect the new contents of FPAC.

Divide Single (FPAC by Memory) (Long Displacement)

LFDMS *fpac,[@]displacement[,index]*



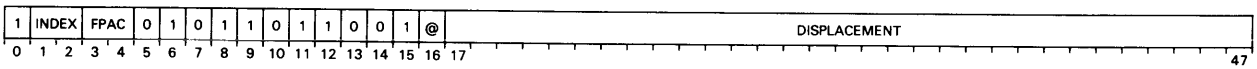
Divides the 32-bit floating-point number in bits 0–31 of FPAC by the 32-bit floating-point number in the source location and places the normalized result in FPAC.

Computes the effective address, *E*. Uses *E* to address a single-precision (double-word) operand. Divides the floating-point number in bits 0–31 of the specified FPAC by this 32-bit floating-point number. Places the normalized result in the specified FPAC. Leaves the contents of the source location unchanged and updates the *Z* and *N* flags in the floating-point status register to reflect the new contents of FPAC.

Sets bits 32–63 of FACD to 0.

Load Floating-Point Double (Long Displacement)

LFLDD *fpac,[@]displacement[,index]*



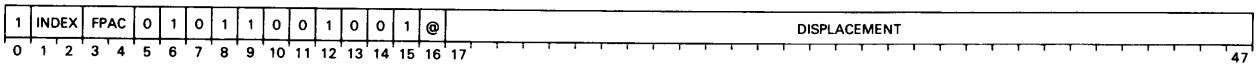
Moves four words out of memory and into a specified FPAC.

Computes the effective address, *E*. Fetches the double-precision floating-point number at the address specified by *E* and places it in FPAC. Updates the *Z* and *N* flags in the FPSR to reflect the new contents of FPAC.

NOTE: *This instruction will move unnormalized data without change, but the* Z *and* N *flags will be undefined.*

Load Floating-Point Single (Long Displacement)

LFLDS *fpac,[@]displacement[,index]*



Moves two words out of memory into a specified FPAC.

Computes the effective address *E*. Fetches the single-precision floating-point number at the address specified by *E*. Places the number in the high-order bits of FPAC. Sets the low-order 32 bits of FPAC to 0. Updates the *Z* and *N* flags in the floating-point status register to reflect the new contents of FPAC.

NOTE: *This instruction will move unnormalized data without change, but the* Z *and* N *flags will be undefined.*

Load Floating-Point Status (Long Displacement)

LFLST *[@]displacemment[,index]*

Moves the contents of four specified memory locations to the floating-point status register.

Computes the effective address, *E*. Places the 64-bit operand addressed by *E* in the floating-point status register as follows:

- Places bits 0–15 of the operand in bits 0–15 of the FPSR. Sets bits 16–32 of the FPSR to 0.
- If *ANY* is 0, bits 33–63 of the FPSR are undefined.
- If *ANY* is 1, the instruction places bits 28–31 of the operand in bits 28–31 (INP) of the FPSR, and places bits 33–63 of the operand in bits 33–63 (FPPC) of the FPSR.

NOTES: *This instruction does not set the* ANY *flag from memory. If any of bits 1–4 are loaded as 1,* ANY *is set to 1; otherwise,* ANY *is 0.*

Bits 12–15 of the FPSR are not set from memory. These bits are the floating-point identification code and cannot be changed. Refer to the specific functional characteristics manual for the code to use.

This instruction initiates a floating-point trap if ANY *and* TE *are both 1 after the FPPC is loaded.*

Multiply Double (FPAC by Memory) (Long Displacement)

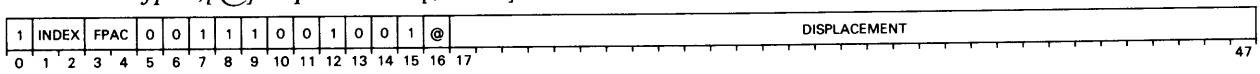
LFMMD *fpac,[@]displacement[,index]*

Multiplies the 64-bit floating-point number in the source location by the 64-bit floating-point number in FPAC and places the normalized result in FPAC.

Computes the effective address, *E*. Uses *E* to address a double-precision (four-word) operand. Multiplies this 64-bit floating-point number by the 64-bit floating-point number in the specified FPAC. Places the normalized result in the specified FPAC. Leaves the contents of the source location unchanged and updates the *Z* and *N* flags in the floating-point status register to reflect the new contents of FPAC.

Multiply Single (FPAC by Memory) (Long Displacement)

LFMMS *fpac,[@]displacement[,index]*

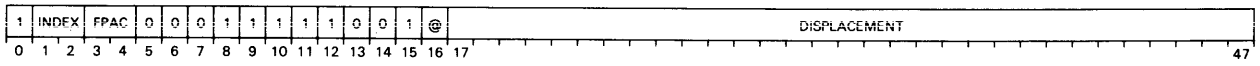


Multiplies the 32-bit floating-point number in the source location by the 32-bit floating-point number in bits 0–31 of FPAC and places the normalized result in FPAC.

Computes the effective address, *E*. Uses *E* to address a single-precision (double-word) operand. Multiplies this 32-bit floating-point number by the floating-point number in bits 0–31 of the specified FPAC. Places the normalized result in bits 0–31 of the specified FPAC. Sets bits 32–63 of FPAC to 0. Leaves the contents of the source location unchanged and updates the *Z* and *N* flags in the floating-point status register to reflect the new contents of FPAC.

Subtract Double (Memory from FPAC) (Long Displacement)

LFSMD *fpac,[@]displacement[,index]*

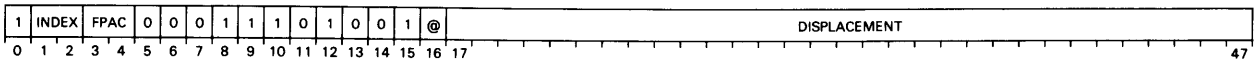


Subtracts the 64-bit floating point number in the source location from the 64-bit floating point number in FPAC and places the normalized result in FPAC.

Computes the effective address, *E*. Uses *E* to address a double-precision (four-word) operand. Subtracts this 64-bit floating-point number from the 64-bit floating-point number in the specified FPAC. Places the normalized result in the specified FPAC. Leaves the contents of the source location unchanged and updates the *Z* and *N* flags in the floating-point status register to reflect the new contents of FPAC.

Subtract Single (Memory from FPAC) (Long Displacement)

LFSMS *fpac,[@]displacement[,index]*

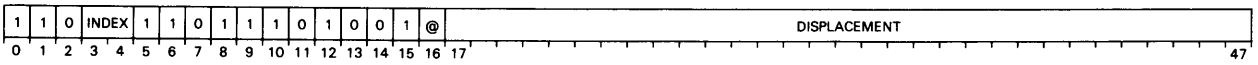


Subtracts the 32-bit floating-point number in the source location from the 32-bit floating-point number in bits 0–31 of FPAC and places the normalized result in FPAC.

Computes the effective address, *E*. Uses *E* to address a single-precision (double word) operand. Subtracts this 32-bit floating-point number from the floating-point number in bits 0–31 of the specified FPAC. Places the normalized result in the specified FPAC. Sets bits 32–63 of FPAC to 0. Leaves the contents of the source location unchanged and updates the *Z* and *N* flags in the floating-point status register to reflect the new contents of FPAC.

Store Floating-Point Status (Long Displacement)

LFSST *[@]displacement[,index]*



Moves the contents of the FPSR to four specified memory locations.

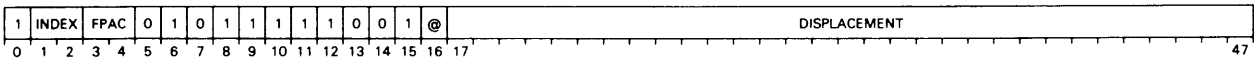
Computes the effective address, *E*, of two sequential, 32-bit locations in memory. Stores the contents of the FPSR in these locations as follows:

- Stores bits 0–15 of the FPSR in the first memory word.
- If *ANY* is 0, sets bits 16–31 of the first memory double word to 0.
- If *ANY* is 1, sets bits 16–27 of the first memory double word to 0, and stores bits 28–31 (INP) of the FPSR into bits 28–31 of the first memory double word.
- Sets bit 0 of the second memory double word to 0.
- If *ANY* is 0, the contents of bits 1–31 of the second memory double word are undefined.
- If *ANY* is 1, the instruction stores bits 33–63 of the FPSR into bits 1–31 of the second memory double word.

NOTE: *This instruction does not initiate a floating-point trap under any conditions of the FPSR.*

Store Floating-Point Double (Long Displacement)

LFSTD *fpac,[@]displacement[,index]*



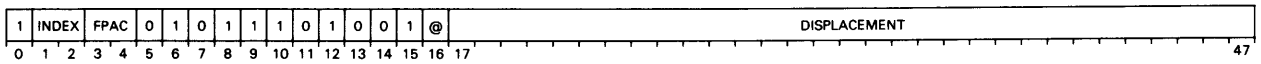
Stores the contents of a specified FPAC into a memory location.

Computes the effective address, *E*. Places the floating-point number contained in FPAC in memory beginning at the location addressed by *E*. Destroys the previous contents of the addressed memory location. The contents of FPAC and the condition codes in the FPSR remain unchanged.

NOTE: *This instruction will move unnormalized data without change.*

Store Floating-Point Single (Long Displacement)

LFSTS *fpac,[@]displacement[,index]*



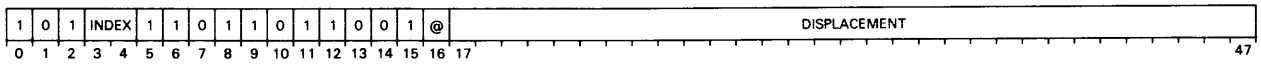
Stores the contents of a specified FPAC into a memory location.

Computes the effective address, *E*. Places the 32 high-order bits of FPAC in memory beginning at the location addressed by *E*. Destroys the previous contents of the addressed memory location. The contents of FPAC and the condition codes in the FPSR remain unchanged.

NOTE: *This instruction will move unnormalized data without change.*

Jump (Long Displacement)

LJMP *[@]displacement[,index]*



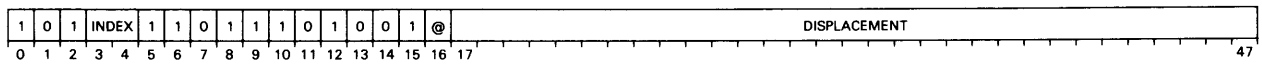
Loads an effective address into the program counter.

Calculates the effective address, *E*. Loads *E* into the PC. Carry is unchanged and *overflow* is 0.

NOTE: *The calculation of* E *is forced to remain within the current segment of execution.*

Jump to Subroutine (Long Displacement)

LJSR *[@]displacement[,index]*



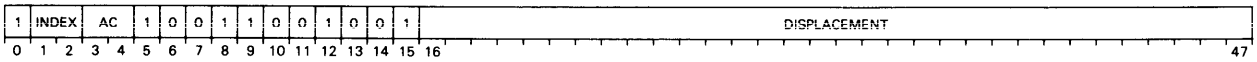
Saves a return address and transfers control to a subroutine.

Calculates the effective address, *E*. Loads AC3 with the current 31-bit value of the program counter plus three. Loads *E* into the PC. Carry is unchanged and *overflow* is 0.

NOTE: *The calculation of* E *is forced to remain within the current segment of execution.*

Load Byte (Long Displacement)

LLDB *ac,displacement[,index]*

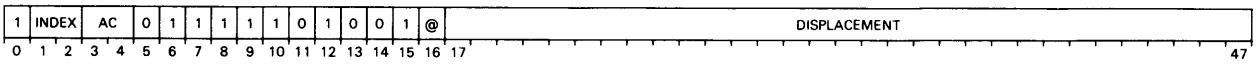


Calculates an effective byte address and loads the byte into the specified accumulator.

Calculates the effective byte address. Uses the byte address to reference a byte in memory. Loads the addressed byte into bits 24–31 of the specified accumulator, then zero extends the value to 32 bits. Carry is unchanged and *overflow* is 0.

Load Effective Address (Long Displacement)

LLEF *ac,[@]displacement[,index]*



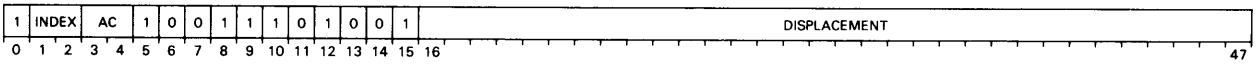
Loads an effective address into an accumulator.

Calculates the effective address, *E*. Loads *E* into the specified accumulator. Carry is unchanged and *overflow* is 0.

NOTE: *Bit 0 of the result is guaranteed to be 0.*

Load Effective Byte Address (Long Displacement)

LLEFB *ac,displacement[,index]*

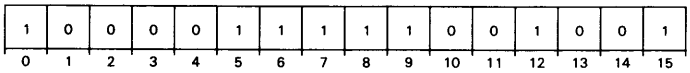


Loads an effective byte address into an accumulator.

Calculates the effective byte address. Loads the byte address into the specified accumulator. Carry is unchanged and *overflow* is 0.

Load Modified and Referenced Bits

LMRF



Loads the modified and referenced bits of a pageframe into AC0.

AC1 contains a pageframe number in bits 13–31.

The bits are loaded into AC0, right-justified, and zero-filled. The modified bit is located in bit 30; the reference bit, in bit 31 of the pageframe. The instruction then resets the referenced bit just accessed to 0. Carry is unchanged and *overflow* is 0.

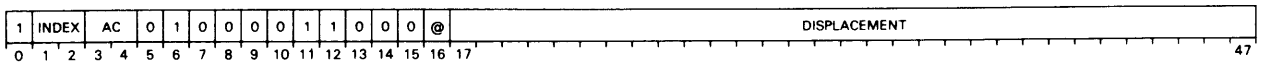
If the address translator is not enabled, undefined results will occur.

Specification of a nonexistent pageframe results in indeterminate data.

NOTE: *This is a privileged instruction.*

Narrow Add Memory Word to Accumulator (Long Displacement)

LNADD *ac,[@]displacement[,index]*

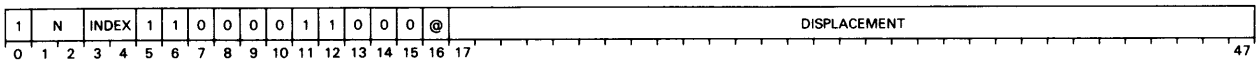


Adds an integer contained in a memory location to an integer contained in an accumulator.

Calculates the effective address, *E*. Adds the 16-bit integer contained in the location specified by *E* to the integer contained in bits 16–31 of the specified accumulator. Sign extends the 16-bit result to 32 bits and loads it into the specified accumulator. Sets carry to the value of ALU carry and *overflow* to 1, if there is an ALU overflow. The contents of the referenced memory location remain unchanged.

Narrow Add Immediate (Long Displacement)

LNADI *n,[@]displacement[,index]*



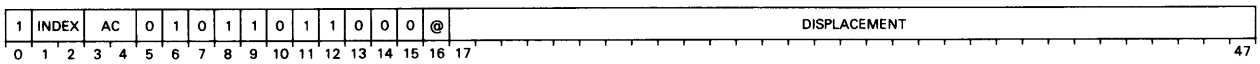
Adds an integer in the range of 1 to 4 to an integer contained in a 16-bit memory location.

Adds the value $n+1$ to the 16-bit contents of the specified memory location, where n is an integer in the range of 0 to 3. Sets carry to the value of ALU carry (16-bit operation). Sets *overflow* to 1, if there is an ALU overflow (16-bit operation).

NOTE: *The assembler takes the coded value of* n *and subtracts one from it before placing it in the immediate field. Therefore, the programmer should code the exact value to be added.*

Narrow Divide Memory Word (Long Displacement)

LNDIV *ac,[@]displacement[,index]*

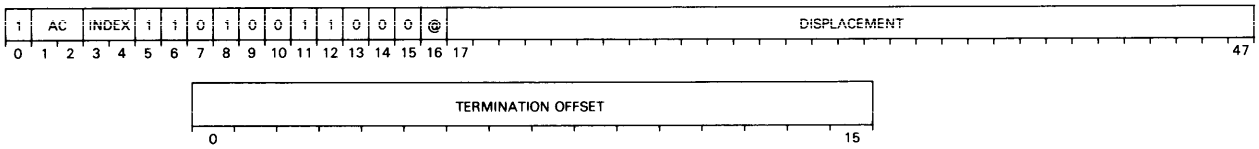


Divides an integer contained in an accumulator by an integer in memory.

Calculates the effective address, *E*. Sign extends the integer contained in bits 16–31 of the specified accumulator to 32 bits and divides it by the 16-bit integer contained in the location specified by *E*. If the quotient is within the range −32,768 to +32,767 inclusive, sign extends the result to 32 bits and loads it into the specified accumulator. If the quotient is outside of this range, or the memory word is zero, the instruction sets *overflow* to 1 and leaves the specified accumulator unchanged. Otherwise, *overflow* is 0. The contents of the referenced memory location and carry remain unchanged.

Narrow Do Until Greater Than (Long Displacement)

LNDO *ac,termination offset, [@]displacement[,index]*



Increments a memory location, compares it to the AC, and takes a normal exit if the location is still less than or equal to the AC.

Increments a 16-bit memory location, sign extends it to 32 bits, and compares it to the AC. If the memory location is greater than the AC, then a PC relative branch is made by adding the termination offset to PC+1. If the memory location is less than or equal to the AC, then the next instruction is executed.

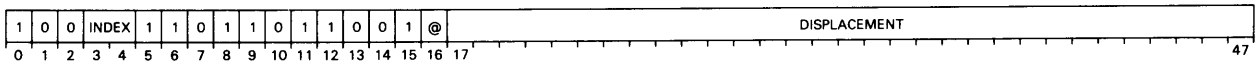
In either case, the instruction loads the incremental memory location into the AC.

If a fixed-point overflow trap occurs while incrementing the DO-loop variable, the contents of the specified memory location and the PC value in the return block are undefined.

Sets carry to the value of ALU carry. Sets *overflow* to 1, if there is an ALU overflow caused by the increment.

Narrow Decrement and Skip if Zero (Long Displacement)

LNDSZ *[@]displacement[,index]*



Decrements the contents of a location and skips the next word if the decremented value is zero.

Calculates the effective address, *E*. Decrements by one the contents of the 16-bit memory location addressed by *E*. If the result is equal to zero, then the instruction skips the next sequential word. Carry is unchanged and *overflow* is 0.

NOTE: *This instruction is indivisible.*

Narrow Increment and Skip if Zero (Long Displacement)

LNISZ *[@]displacement[,index]*

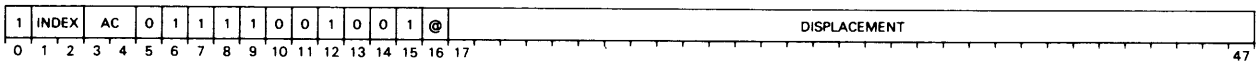
Increments the contents of a location and skips the next word if the incremented value is zero.

Calculates the effective address, *E*. Increments by one the contents of the 16-bit memory location addressed by *E*. If the result is equal to zero, then the instruction skips the next sequential word. Carry is unchanged and *overflow* is 0.

NOTE: *This instruction is indivisible.*

Narrow Load Accumulator (Long Displacement)

LNLDA *ac,[@]displacement[,index]*

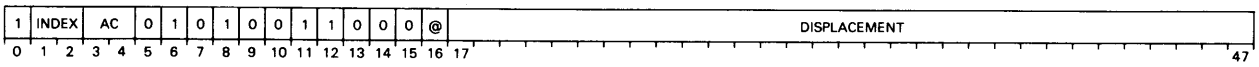


Sign extends and loads the contents of a memory location into an accumulator.

Calculates the effective address, *E*. Fetches the 16-bit fixed-point integer contained in the location specified by *E*. Sign extends this integer to 32 bits and loads it into the specified accumulator. Carry is unchanged and *overflow* is 0.

Narrow Multiply Memory Word (Long Displacement)

LNMUL *ac,[@]displacement[,index]*

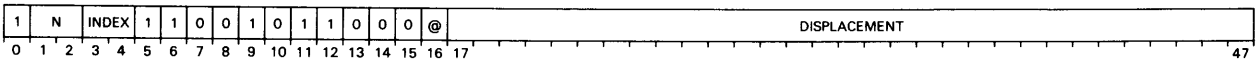


Multiplies an integer in memory by an integer in an accumulator.

Calculates the effective address, *E*. Multiplies the 16-bit, signed integer contained in the location referenced by *E* by the signed integer contained in bits 16–31 of the specified accumulator. If the result is outside the range of −32,768 to +32,767 inclusive, sets *overflow* to 1; otherwise, *overflow* is 0. Sign extends the result to 32 bits and places the result in the specified accumulator. The contents of the referenced memory location and carry remain unchanged.

Narrow Subtract Immediate (Long Displacement)

LNSBI *n,[@]displacement[,index]*



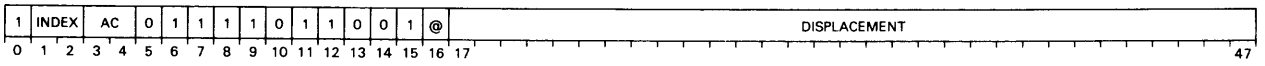
Subtracts an integer in the range of 1 to 4 from an integer contained in a 16-bit memory location.

Subtracts the value $n+1$ from the 16-bit value contained in the specified memory location, where *n* is an integer in the range of 0 to 3. Sets carry to the value of ALU carry (16-bit operation). Sets *overflow* to 1, if there is an ALU overflow (16-bit operation).

NOTE: *The assembler takes the coded value of* n *and subtracts one from it before placing it in the immediate field. Therefore, the programmer should code the exact value to be subtracted.*

Narrow Store Accumulator (Long Displacement)

LNSTA *ac,[@]displacement[,index]*

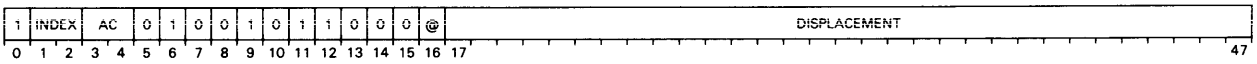


Stores the contents of an accumulator in a memory location.

Calculates the effective address, *E*. Stores a copy of the low-order 16 bits of the specified accumulator in the memory location specified by *E*. Carry is unchanged; *overflow* is 0.

Narrow Subtract Memory Word (Long Displacement)

LNSUB *ac,[@]displacement[,index]*

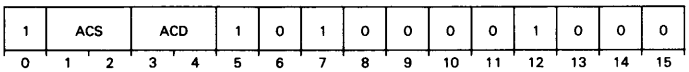


Subtracts an integer in memory from an integer in an accumulator.

Calculates the effective address, *E*. Subtracts the 16-bit integer contained in the location referenced by *E* from the integer contained in bits 16–31 of the specified accumulator. Sign extends the result to 32 bits and stores it in the specified accumulator. Sets carry to the value of ALU carry and *overflow* to 1, if there is an ALU overflow. The contents of the specified memory location remain unchanged.

Locate Lead Bit

LOB *acs,acd*



Counts and adds the number of high-order zeroes in an accumulator to another accumulator.

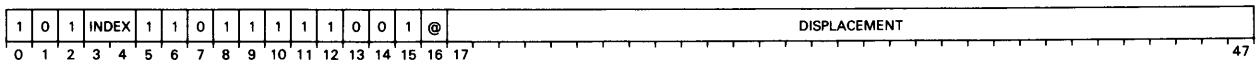
Adds a number equal to the number of high-order zeroes in the contents of bits 16–31 of ACS to the signed, 16-bit, two's complement number contained in bits 16–31 of ACD. The contents of ACS and the state of carry remain unchanged. *Overflow* is 0.

Bits 0–15 of the modified accumulator are undefined after completion of this instruction.

NOTE: *If ACS and ACD are specified to be the same accumulator, the instruction functions as described above, except that the contents of ACS will be changed.*

Push Address (Long Displacement)

LPEF *[@]displacement[,index]*



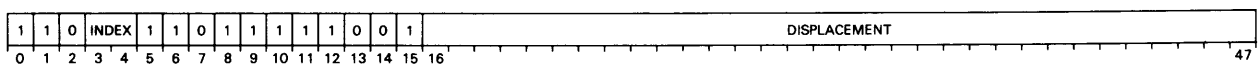
Pushes an address onto the wide stack.

Calculates the effective address, *E*. Pushes *E* onto the wide stack, then checks for stack overflow. Carry is unchanged and *overflow* is 0.

NOTE: *Bit 0 of the pushed address is guaranteed to be 0.*

Push Byte Address (Long Displacement)

LPEFB *displacement[,index]*

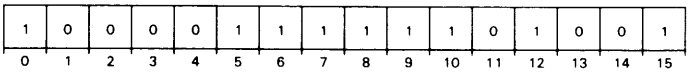


Pushes a byte address onto the wide stack.

Calculates a 32-bit byte address. Pushes this byte address onto the wide stack, then checks for stack overflow. Carry is unchanged and *overflow* is 0.

Load Physical And Conditional Skip

LPHY



Translates the logical address contained in AC1 to a physical address.

AC1 contains a logical word address.

If the address translator is disabled, this instruction does nothing. The next word is executed.

If the address translator is enabled, then the actions described below occur.

The instruction compares the ring field of AC1 to the current ring. If AC1's ring field is less than the current ring field, then a protection fault (AC1 = 4) occurs.

If AC1's ring field is greater than or equal to the current ring, then the instruction references the SBR specified by AC1. If the SBR contents are invalid or a depth fault is indicated, then the instruction ends and the next instruction is executed. The contents of AC0 will be unchanged.

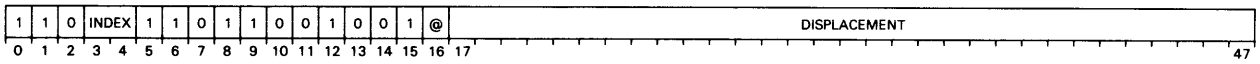
If the contents of the SBR are valid and no depth fault is indicated, the instruction loads AC0 with the last resident PTE. If the PTE indicates no page or validity faults, the instruction loads AC2 with the 32-bit physical word address of the logical address contained in AC1. The next sequential word is skipped.

If the PTE signals a page or validity fault, the contents of AC2 remain unchanged. The next sequential word is executed.

The instruction leaves carry unchanged; *overflow* is 0.

Push Jump (Long Displacement)

LPSHJ *[@]displacement[,index]*



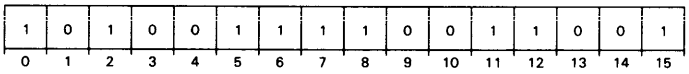
Saves a return address on the wide stack and jumps to a specified location.

Calculates the effective address, *E*. Pushes the current 31-bit value of the program counter plus three onto the wide stack. Loads the PC with *E*. Sequential operation continues with the word addressed by the updated value of the program counter. Carry is unchanged and *overflow* is 0.

NOTE: *The value pushed onto the wide stack will always point to a location in the current ring.*

Load Processor Status Register into AC0

LPSR

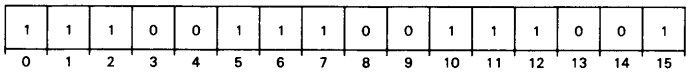


Loads the contents of the PSR into AC0.

Loads the contents of the defined bits of the PSR into the corresponding bits of AC0. Fills the rest of AC0 with zeroes. The contents of the PSR remain unchanged. Carry is unchanged and *overflow* is 0.

Load Page Table Entry

LPTE



Obtains the Page Table Entry (PTE) that corresponds to the translation of a given logical address.

AC1 contains a logical address whose PTE is desired. AC3 contains a pointer to a table of eight double words. Each double word contains a Segment Base Register (SBR). The addresses in AC1 and AC3 may not specify indirection.

The contents of AC1 bits 1 through 3 (the ring bits) are used to index by double words into the SBR table pointed to by AC3 to obtain an SBR for use in the translation.

If no exceptions are encountered, the next sequential word is skipped and execution continues with the PTE returned to AC0 and its physical address to AC2. AC1 and AC3 are unmodified.

If an exception does occur, execution continues with the next sequential instruction and no accumulators are modified.

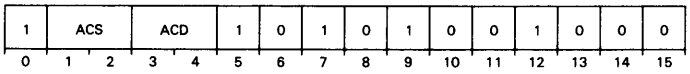
Exceptions which will take the error path (in order of detection) are: invalid SBR, Page Table depth error, invalid second level PTE, non-resident first level PTE.

NOTE: *The following conditions are not considered errors: invalid first level PTE and non-resident object page.*

This is a privileged instruction.

Locate and Reset Lead Bit

LRB *acs,acd*



Performs a *Locate Lead Bit* instruction and sets the lead bit to 0.

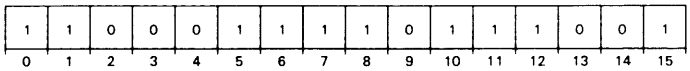
Adds a number equal to the number of high-order zeroes in the contents of bits 16–31 of ACS to the signed, 16-bit, two's complement number contained in bits 16–31 of ACD. Sets the leading 1 in bits 16–31 of ACS to 0. Carry remains unchanged; *overflow* is 0.

Bits 0–15 of the modified accumulator are undefined after completion of this instruction.

NOTE: *If ACS and ACD are specified to be the same accumulator, then the instruction sets the leading 1 in that accumulator to 0 and no count is taken.*

Load All Segment Base Registers

LSBRA



Loads the eight SBRs with new values.

AC0 contains the starting address of an 8-double-word block.

The instruction loads a copy of the contents of these words into the SBRs as shown in the following list.

Double Word in Block	Destination	Order Moved
1	SBR0	First
2	SBR1	Second
3	SBR2	Third
4	SBR3	Fourth
5	SBR4	Fifth
6	SBR5	Sixth
7	SBR6	Seventh
8	SBR7	Eighth

After loading the SBRs, the instruction purges the address translator. If the address translator was disabled at the beginning of this instruction cycle, the processor enables it now.

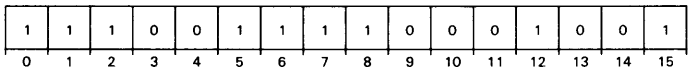
If an invalid address is loaded into SBR0, the processor disables the address translator and a protection fault occurs (code = 3 in AC1). This means that logical addresses are identical to physical addresses and the fault is processed in physical address space.

The instruction leaves AC0 and carry unchanged; *overflow* is 0.

NOTE: *This is a privileged instruction.*

Load Segment Base Registers 1–7

LSBRS



Loads SBR1 through SBR7 with new values.

AC0 contains the starting address of a block of seven double words. The list gives how the instruction loads a copy of the contents of these words into the SBRs.

Double Word in Block	Destination	Order Moved
1	SBR1	First
2	SBR2	Second
3	SBR3	Third
4	SBR4	Fourth
5	SBR5	Fifth
6	SBR6	Sixth
7	SBR7	Seventh

After loading the SBRs, the instruction purges the address translator. If the address translator was disabled at the beginning of this instruction cycle, the processor enables it now.

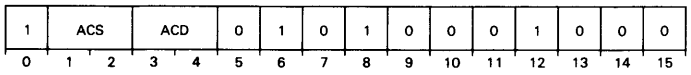
If SBR0 contains invalid information, then the processor disables the address translator and a protection fault occurs (code = 3 in AC1). This means that logical addresses are identical to physical addresses and the fault is processed in physical address space.

The instruction leaves AC0 and carry unchanged; *overflow* is 0.

NOTE: *This is a privileged instruction.*

Logical Shift

LSH *acs,acd*



Shifts the contents of a 16-bit accumulator left or right, depending on the contents of another accumulator.

Shifts the contents of bits 16–31 of ACD either left or right depending on the number contained in bits 24–31 of ACS. The signed, 8-bit two's complement number contained in bits 24–31 of ACS determines the direction of the shift and the number of bits to be shifted. If the number in bits 24–31 of ACS is positive, shifting is to the left; if the number in bits 24–31 of ACS is negative, shifting is to the right. If the number in bits 24–31 of ACS is zero, no shifting is performed. Bits 16–23 of ACS are ignored.

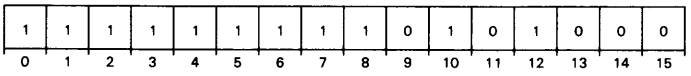
The number of bits shifted is equal to the magnitude of the number in bits 24–31 of ACS. Bits shifted out are lost and the vacated bit positions are filled with zeroes. Carry and the contents of ACS remain unchanged. *Overflow* is 0.

Bits 0–15 of the modified accumulator are undefined after completion of this instruction.

NOTE: *If the magnitude of the number in bits 24–31 of ACS is greater than 15, all bits of ACD are set to 0. Carry and the contents of ACS remain unchanged.*

Load Sign

LSN



Under the control of accumulators AC1 and AC3, evaluates a decimal number in memory and returns in AC1 a code that classifies the number as zero or nonzero and identifies its sign.

The meaning of the returned code is as follows:

Value of Number	Code
Positive nonzero	+1
Negative nonzero	−1
Positive zero	0
Negative zero	−2

Bits 16–31 of AC1 must contain the data type indicator describing the number.

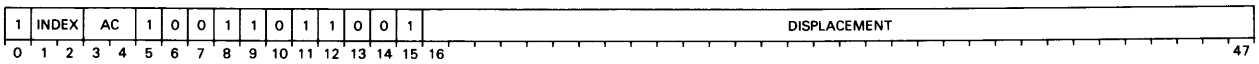
Bits 16–31 of AC3 must contain a byte pointer which is the address of the high-order byte of the number.

Upon successful termination, the contents of AC0 remain unchanged; AC1 contains the value code; AC2 contains the original contents of AC3; and the contents of AC3 are unpredictable. Carry remains unchanged. The contents of the addressed memory locations remain unchanged. *Overflow* is 0.

The 31-bit effective address generated by this instruction is constrained to be within the first 64 Kbytes of the current segment.

Store Byte (Long Displacement)

LSTB *ac,displacement[,index]*

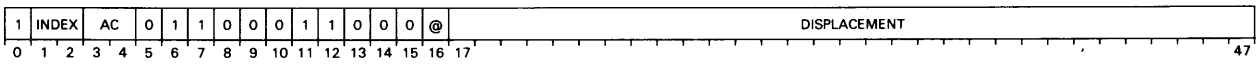


Stores the low-order byte of the specified accumulator in memory.

Calculates the effective byte address. Moves a copy of the contents of bits 24–31 of the specified accumulator into memory at the location specified by the byte address. Carry is unchanged and *overflow* is 0.

Wide Add Memory Word to Accumulator (Long Displacement)

LWADD *ac,[@]displacement[,index]*

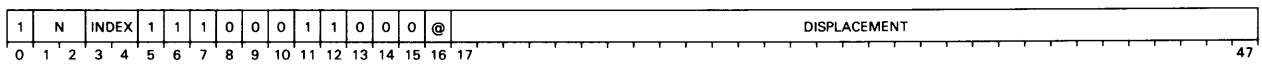


Adds an integer contained in memory to an integer contained in an accumulator.

Calculates the effective address, *E*. Adds the 32-bit integer contained in the location specified by *E* to the 32-bit integer contained in the specified accumulator. Loads the result into the specified accumulator. Sets carry to the value of ALU carry and *overflow* to 1, if there is an ALU overflow. The contents of the referenced memory location remain unchanged.

Wide Add Immediate (Long Displacement)

LWADI *n,[@]displacement[,index]*



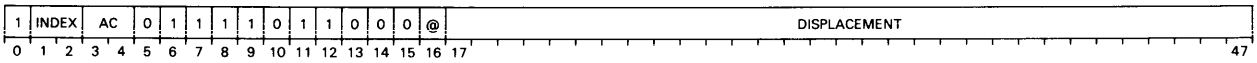
Adds an integer in the range of 1 to 4 to an integer contained in a 32-bit memory location.

Adds the value $n+1$ to the 32-bit fixed-point integer contained in a memory location, where n is an integer in the range of 0 to 3. Sets carry to the value of ALU carry. Sets *overflow* to 1, if there is an ALU overflow.

NOTE: *The assembler takes the coded value of* n *and subtracts one from it before placing it in the immediate field. Therefore, the programmer should code the exact value to be added.*

Wide Divide Memory Word (Long Displacement)

LWDIV *ac,[@]displacement[,index]*



Divides an integer in an accumulator by an integer in memory.

Calculates the effective address, *E*. Sign extends the 32-bit integer contained in the specified accumulator to 64 bits and divides it by the 32-bit integer contained in the location specified by *E*.

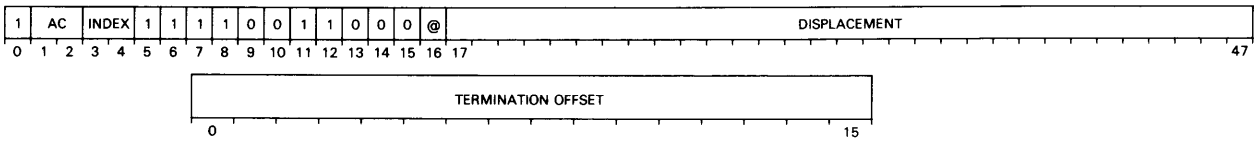
If the quotient is within the range of −2,147,483,648 to +2,147,483,647 inclusive, or if the memory word is zero, the instruction loads the quotient into the specified accumulator. *Overflow* is 0.

If the quotient is outside this range, or if the word in memory is zero, the instruction sets *overflow* to 1 and leaves the specified accumulator unchanged.

The contents of the referenced memory location and carry remain unchanged.

Wide Do Until Greater Than (Long Displacement)

LWDO *ac,termination offset[@]displacement[index]*



Increments a memory location, compares it to the AC, and takes a normal exit if the location is still less than or equal to the AC.

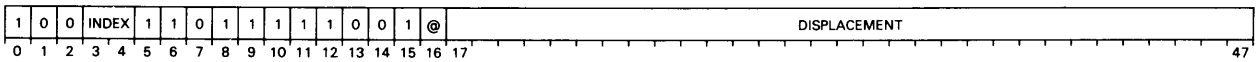
Increments a 32-bit memory location and compares it to the AC. If the memory location is greater than the AC, then a PC relative branch is made by adding the termination offset to PC+1. If the memory location is less than or equal to the AC, then the next instruction is executed. In either case, the instruction loads the incremented memory word into the AC.

If a fixed-point overflow trap occurs while incrementing the DO-loop variable, the contents of the specified memory location and the PC value in the return block are undefined.

Sets carry to the value of ALU carry. Sets *overflow* to 1, if there is an ALU overflow caused by the increment.

Wide Decrement and Skip if Zero (Long Displacement)

LWDSZ *[@]displacement[,index]*



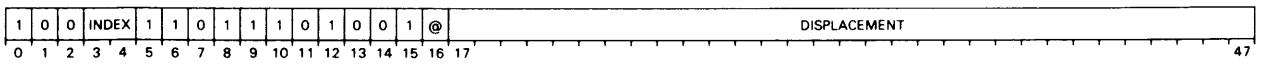
Decrements the contents of a location and skips the next word if the decremented value is zero.

Calculates the effective address, *E*. Decrements by one the contents of the 32-bit memory location addressed by *E*. If the result is equal to zero, then the instruction skips the next sequential word. Carry is unchanged and *overflow* is 0.

NOTE: *This instruction executes in one indivisible memory cycle if the word to be decremented is located on a double-word boundary.*

Wide Increment and Skip if Zero (Long Displacement)

LWISZ *[@]displacement[,index]*



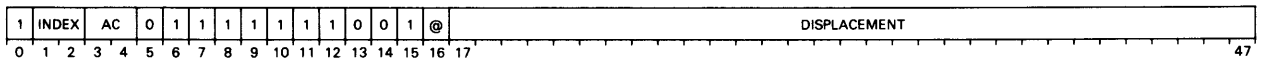
Increments the contents of a location and skips the next word if the incremented value is zero.

Calculates the effective address, *E*. Increments by one the contents of the 32-bit memory location addressed by *E*. If the result is equal to zero, then the instruction skips the next sequential word. Carry is unchanged and *overflow* is 0.

> **NOTE:** *This instruction executes in one indivisible memory cycle if the word to be incremented is located on a double-word boundary.*

Wide Load Accumulator (Long Displacement)

LWLDA *ac,[@]displacement[,index]*

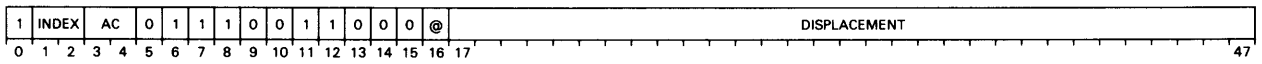


Loads the contents of a memory location into an accumulator.

Calculates the effective address, *E*. Fetches the 32-bit fixed-point integer contained in the location specified by *E*. Loads a copy of this integer into the specified accumulator. Carry is unchanged and *overflow* is 0.

Wide Multiply Memory Word (Long Displacement)

LWMUL *ac,[@]displacement[,index]*



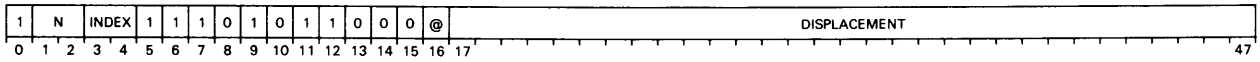
Multiplies an integer in an accumulator by an integer in memory.

Calculates the effective address, *E*. Multiplies the 32-bit, signed integer contained in the location referenced by *E* by the 32-bit, signed integer contained in the specified accumulator. Loads the 32 least significant bits of the result into the specified accumulator.

If the result is outside the range of −2,147,483,648 to +2,147,483,647 inclusive, the instruction sets *overflow* to 1; otherwise, *overflow* is 0. The contents of the referenced memory location and carry remain unchanged.

Wide Subtract Immediate (Long Displacement)

LWSBI *n,[@]displacement[,index]*



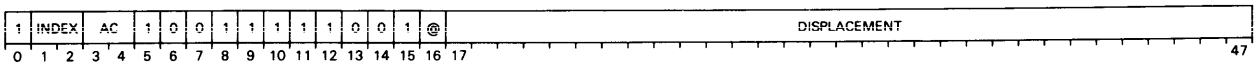
Subtracts an integer in the range of 1 to 4 to an integer contained in a 32-bit memory location.

Subtracts the value $n+1$ from the value contained in the specified 32-bit memory location, where *n* is an integer in the range of 0 to 3. Sets carry to the value of ALU carry. Sets *overflow* to 1, if there is an ALU overflow.

> **NOTE:** *The assembler takes the coded value of* n *and subtracts one from it before placing it in the immediate field. Therefore, the programmer should code the exact value to be subtracted.*

Wide Store Accumulator (Long Displacement)

LWSTA *ac,[@]displacement[,index]*

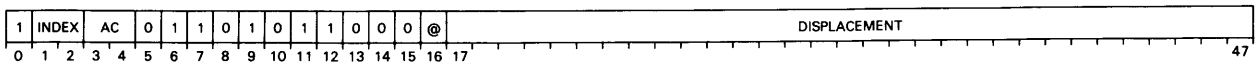


Stores the contents of an accumulator in a memory location.

Calculates the effective address, *E*. Stores a copy of the 32-bit contents of the specified accumulator in the memory location specified by *E*. Carry is unchanged; *overflow* is 0.

Wide Subtract Memory Word (Long Displacement)

LWSUB *ac,[@]displacement[,index]*

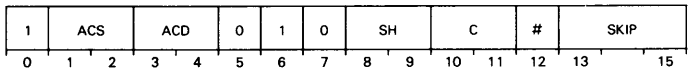


Subtracts an integer in memory from an integer in an accumulator.

Calculates the effective address, *E*. Subtracts the 32-bit integer contained in the memory location referenced by *E* from the 32-bit integer contained in the specified accumulator. Loads the result into the specified accumulator. Sets carry to the value of ALU carry and *overflow* to 1, if there is an ALU overflow. The contents of the specified memory location remain unchanged.

Move

MOV*[c][sh][#]* *acs,acd[,skip]*



Moves the contents of bits 16–31 of an accumulator into another accumulator.

Initializes carry to the specified value. Places the contents of bits 16–31 of ACS in the shifter. Performs the specified shift operation and loads the result of the shift into bits 16–31 of ACD if the no-load bit is 0. If the skip condition is true, the instruction skips the next sequential word. *Overflow* is 0.

If the load option is specified, bits 0–15 of ACD are undefined.

[c]

The processor determines the effect of the CARRY flag (c) on the old value of CARRY before performing the operation (opcode). The following list gives the values of c, bits 10 and 11, and the operation.

Symbol *[c]*	Bits 10-11	Operation
omitted	0 0	Leave CARRY unchanged
Z	0 1	Initialize CARRY to 0
O	1 0	Initialize CARRY to 1
C	1 1	Complement CARRY

[sh]

The processor shifts the CARRY flag and the 16 data bits after performing the instruction operation. The processor can shift the bits left or right one bit position, or it can swap the two bytes. The following list gives the values of sh, bits 8 and 9, and the shift operation.

Symbol *[sh]*	Bits 8-9	Shift Operation
omitted	0 0	Do not shift the result
L	0 1	Shift left
R	1 0	Shift right
S	1 1	Swap the two 8-bit bytes

[#]

Unless you use the no-load option (#), the processor loads the result of the shift operation into the destination accumulator. The no-load option is useful to test the result of the instruction operation without destroying the destination accumulator contents. The following list gives the values of the no-load option, bit 12, and the operation.

Symbol *[#]*	Bit 12	Operation
omitted	0	Load the result into ACD
#	1	Do not load the result and restore the CARRY flag

NOTE: *Do not specify an instruction with the no-load option (#) in combination with either the never skip or always skip option. Thus, the instruction may not end in 1000_2 or 1001_2, other instructions use the bit combinations.*

[skip]

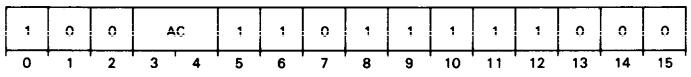
The processor can skip the next instruction if the condition test is true. The following list gives the test conditions, bits 13 to 15, and the operation.

Symbol *[skip]*	Bits 13-15	Operation
omitted	0 0 0	Never skip
SKP	0 0 1	Always skip
SZC	0 1 0	Skip if CARRY is 0
SNC	0 1 1	Skip if CARRY is not 0
SZR	1 0 0	Skip if the result is 0
SNR	1 0 1	Skip if the result is not 0
SEZ	1 1 0	Skip if either CARRY or the result is 0
SBN	1 1 1	Skip if both CARRY and the result are not 0

When the instruction performs a skip, it skips the next sequential 16-bit word. Make sure that a skip does not transfer control to the middle of a 32-bit or longer instruction.

Modify Stack Pointer

MSP *ac*



Changes the value of the stack pointer and tests for potential overflow.

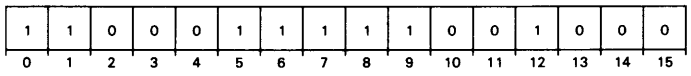
Adds the signed two's-complement number in bits 16–31 of the specified accumulator to the value of the stack pointer and places the result in location 40_8. The instruction then checks for overflow by comparing the result in location 40_8 with the value of the stack limit. If the result in location 40_8 is less than or equal to the stack limit, then the instruction ends.

If the result is greater than the stack limit, the instruction changes the value of location 40 back to its original contents before the add. The instruction pushes a standard return block. The program counter in the return block contains the address of the *Modify Stack Pointer* instruction.

After pushing the return block, the program counter contains the address of the stack fault routine. The stack pointer is updated with the value used to push the return block, and control transfers to the stack fault routine. Carry remains unchanged and *overflow* is 0.

Unsigned Multiply

MUL



Multiplies the unsigned contents of two accumulators and adds the result to the unsigned contents of a third accumulator. The result is an unsigned 32-bit integer in two accumulators.

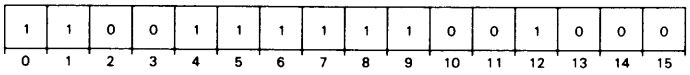
Multiplies the unsigned, 16-bit number in bits 16–31 of AC1 by the unsigned, 16-bit number in bits 16–31 of AC2 to yield an unsigned, 32-bit intermediate result. The unsigned, 16-bit number in bits 16–31 of AC0 is added to the intermediate result to produce the final result. The final result is an unsigned, 32-bit number and occupies bits 16–31 of both AC0 and AC1. Bit 16 of AC0 is the high-order bit of the result and bit 31 of AC1 is the low-order bit. The contents of AC2 remain unchanged.

Because the result is a double-length number, overflow cannot occur. Carry remains unchanged and *overflow* is 0.

Bits 0–15 of the modified accumulator are undefined after completion of this instruction.

Signed Multiply

MULS



Multiplies the signed contents of two accumulators and adds the result to the signed contents of a third accumulator. The result is a signed 32-bit integer in two accumulators.

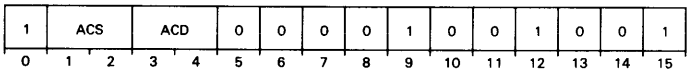
Multiplies the signed, 16-bit two's complement number in bits 16–31 of AC1 by the signed, 16-bit two's complement number in bits 16–31 of AC2 to yield a signed, 32-bit two's complement intermediate result. The signed, 16-bit two's complement number in bits 16–31 of AC0 is added to the intermediate result to produce the final result. The final result is a signed, 32-bit two's complement number which occupies bits 16–31 of both AC0 and AC1. Bit 16 of AC0 is the sign bit of the result and bit 31 of AC1 is the low-order bit. The contents of AC2 remain unchanged.

Because the result is a double-length number, overflow cannot occur. Carry remains unchanged and *overflow* is 0.

Bits 0–15 of the modified accumulator are undefined after completion of this instruction.

Narrow Add

NADD *acs,acd*

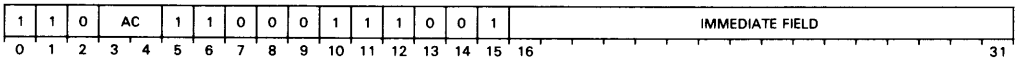


Adds two integers contained in accumulators.

The instruction adds the 16-bit integer contained in bits 16–31 of ACS to the 16-bit integer contained in bits 16–31 of ACD. Stores the result in bits 16–31 of ACD. Sign extends ACD to 32 bits. Sets carry to the value of ALU carry (16-bit operation). If there is an ALU overflow (16-bit operation), NADD sets *overflow* to 1.

Narrow Extended Add Immediate

NADDI *i,ac*

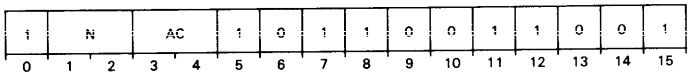


Adds an integer contained in an immediate field to an integer in an accumulator.

Adds the 16-bit value contained in the immediate field to bits 16–31 of the specified accumulator. Stores the result in the lower 16 bits of ACD. Sign extends ACD to 32 bits. Sets carry to the value of ALU carry (16-bit operation). If there is an ALU overflow (16-bit operation), NADDI sets *overflow* to 1.

Narrow Add Immediate

NADI *n,ac*



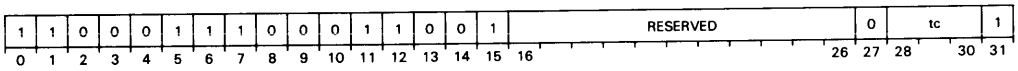
Adds an integer in the range of 1 to 4 to an integer contained in an accumulator.

The instruction adds the value $n+1$ to the 16-bit contents of the specified accumulator, where n is an integer in the range of 0 to 3. Stores the result in the lower 16 bits of the specified accumulator. Sign extends the specified accumulator to 32 bits. Sets carry to the value of ALU carry (16-bit operation). Sets *overflow* to 1 if there is an ALU overflow (16-bit operation).

NOTE: *The assembler takes the coded value of n and subtracts 1 from it before placing it in the immediate field. Therefore, the programmer should code the exact value to be added.*

Narrow Backward Search Queue and Skip

NBStc



The NBStc instruction searches backward through a 16-bit data field. If the search is successful, it skips the next two single-word instructions.

AC1 and AC3 identify a location in a data field as a beginning data element in the queue search. AC1 contains an effective address. AC3 contains a 16-bit word offset (a signed integer in bits 16-31 of AC3). The processor locates the beginning data element by calculating the effective address and adding the word offset.

Bits 16-31 of the top word on the wide stack contain the mask word. The bits in the mask word identify the test location bits to sample.

Bits 28-30 of the NBStc instruction specify the search condition.

tc Value	Bits 28-30 Encoding *	Meaning
SS	0 0 0	Some of the sampled test location bits are 1
SC	0 0 1	Some of the sampled test location bits are 0
AS	0 1 0	All of the sampled test location bits are 1
AC	0 1 1	All of the sampled test location bits are 0
E	1 0 0	The mask and test location are equal
GE	1 0 1	The mask is greater than or equal to the test locaiton
LE	1 1 0	The mask is less than or equal to the test location
NE	1 1 1	The mask and test location are not equal

*The instruction treats the values contained in the mask and in the test location as unsigned integers for the E, GE, LE, and NE test conditions.

The search begins with the addressed data element, and compares it with the 16-bit mask word. The search continues until the processor reaches either the head of the queue or the data element that meets the test condition.

If the search is successful, AC1 contains the effective address of the data element, and the processor then skips the next two single-word instructions. The processor will not

honor interrupts between the time that it completes a successful search and executes the PC+4 instructions.

NOTE: *After a successful search terminates, a new beginning pointer must be placed in AC1 if the search is to be continued through the remainder of the queue.*

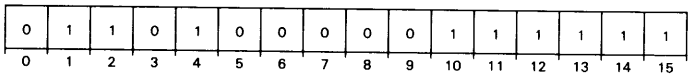
If the search fails, AC1 contains the effective address of the last data element searched, and the processor then executes the next instruction. The processor will honor interrupts between the time that it completes an unsuccessful search and executes the PC+2 instruction.

If the processor interrupts the search, AC1 contains the effective address of the next data element to examine, and the processor then skips the next instruction. The processor will honor interrupts between the time that the interrupt occurs and the processor executes the PC+3 instruction.

For all returns, the contents of CARRY, OVR, WSP, AC0, AC2, and AC3 remain unchanged.

Narrow Load CPU Identification

NCLID

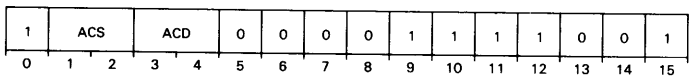


Loads CPU identification into the accumulators. Refer to the specific functional characteristic manual for the accumulator format. Carry is unchanged and overflow is 0.

NOTE: *This instruction can be executed only with Lef mode and I/O protection disabled.*

Narrow Divide

NDIV *acs,acd*

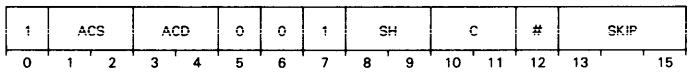


Divides an integer in an accumulator by an integer in another accumulator.

Sign extends bits 16–31 of ACD to 32 bits. Divides this signed integer by the 16-bit signed integer contained in bits 16–31 of ACS. If the quotient is within the range of −32,768 to +32,767 inclusive, sign extends the lower 16 bits of the result to 32 bits and places these 16 bits in ACD. If the quotient is outside of this range, or if ACS is zero, the instruction sets *overflow* to 1 and leaves ACD unchanged. Otherwise, *overflow* is 0. The contents of ACS and carry always remain unchanged.

Negate

NEG*[c][sh][#] acs,acd[,skip]*



Forms the two's complement of the contents of bits 16–31 of an accumulator.

Initializes carry to the specified value. Places the two's complement of the unsigned, 16-bit number in bits 16–31 of ACS in the shifter. If the negate operation produces a carry of 1 out of the high-order bit, the instruction complements carry. Performs the specified shift operation and places the result in bits 16–31 of ACD if the no-load bit is 0. If the skip condition is true, the instruction skips the next sequential word. *Overflow* is 0.

If the load option is specified, bits 0–15 of ACD are undefined.

NOTE: *If ACS contains 0, the instruction complements carry.*

[c]

The processor determines the effect of the CARRY flag (c) on the old value of CARRY before performing the operation (opcode). The following list gives the values of c, bits 10 and 11, and the operation.

Symbol *[c]*	Bits 10-11	Operation
omitted	0 0	Leave CARRY unchanged
Z	0 1	Initialize CARRY to 0
O	1 0	Initialize CARRY to 1
C	1 1	Complement CARRY

[sh]

The processor shifts the CARRY flag and the 16 data bits after performing the instruction operation. The processor can shift the bits left or right one bit position, or it can swap the two bytes. The following list gives the values of sh, bits 8 and 9, and the shift operation.

Symbol *[sh]*	Bits 8-9	Shift Operation
omitted	0 0	Do not shift the result
L	0 1	Shift left
R	1 0	Shift right
S	1 1	Swap the two 8-bit bytes

[#]

Unless you use the no-load option (#), the processor loads the result of the shift operation into the destination accumulator. The no-load option is useful to test the result of the instruction operation without destroying the destination accumulator contents. The following list gives the values of the no-load option, bit 12, and the operation.

Symbol *[#]*	Bit 12	Operation
omitted	0	Load the result into ACD
#	1	Do not load the result and restore the CARRY flag

NOTE: *Do not specify an insruction with the no-load option (#) in combination with either the never skip or always skip option. Thus, the instruction may not end in 1000_2 or 1001_2, other instructions use the bit combinations.*

[skip]

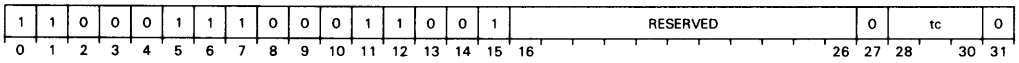
The processor can skip the next instruction if the condition test is true. The following list gives the test conditions, bits 13 to 15, and the operation.

Symbol *[skip]*	Bits 13-15	Operation
omitted	0 0 0	Never skip
SKP	0 0 1	Always skip
SZC	0 1 0	Skip if CARRY is 0
SNC	0 1 1	Skip if CARRY is not 0
SZR	1 0 0	Skip if the result is 0
SNR	1 0 1	Skip if the result is not 0
SEZ	1 1 0	Skip if either CARRY or the result is 0
SBN	1 1 1	Skip if both CARRY and the result are not 0

When the instruction performs a skip, it skips the next sequential 16-bit word. Make sure that a skip does not transfer control to the middle of a 32-bit or longer instruction.

Narrow Forward Search Queue and Skip

NFStc



The NFStc instruction searches forward through a 16-bit data field. If the search is successful, it skips the next two single-word instructions.

AC1 and AC3 identify a location in a data field as a beginning data element in the queue search. AC1 contains an effective address. AC3 contains a 16-bit word offset (a signed integer in bits 16-31 of AC3). The processor locates the beginning data element by calculating the effective address and adding the word offset.

Bits 16-31 of the top word on the wide stack contain the mask word. The bits in the mask word identify the test location bits to sample.

Bits 28-30 of the NFStc instruction specify the search condition.

tc Value	Bits 28-30 Encoding *	Meaning
SS	0 0 0	Some of the sampled test location bits are 1
SC	0 0 1	Some of the sampled test location bits are 0
AS	0 1 0	All of the sampled test location bits are 1
AC	0 1 1	All of the sampled test location bits are 0
E	1 0 0	The mask and test location are equal
GE	1 0 1	The mask is greater than or equal to the test location
LE	1 1 0	The mask is less than or equal to the test location
NE	1 1 1	The mask and test location are not equal

*The instruction treats the values contained in the mask and in the test location as unsigned integers for the E, GE, LE, and NE test conditions.

The search begins with the addressed data element, and compares it with the 16-bit mask word. The search continues until the processor reaches either the tail of the queue or the data element that meets the test condition.

If the search is successful, AC1 contains the effective address of the data element, and the processor then skips the next two single-word instructions. The processor will not honor interrupts between the time that it completes a successful search and executes the PC+4 instructions.

NOTE: *After a successful search terminates, a new beginning pointer must be placed in AC1 if the search is to be continued through the remainder of the queue.*

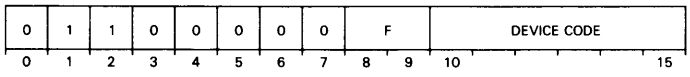
If the search fails, AC1 contains the effective address of the last data element searched, and the processor then executes the next instruction. The processor will honor interrupts between the time that it completes an unsuccessful search and executes the PC+2 instruction.

If the processor interrupts the search, AC1 contains the effective address of the next data element to examine, and the processor then skips the next instruction. The processor will honor interrupts between the time that the interrupt occurs and the processor executes the PC+3 instruction.

For all returns, the contents of CARRY, OVR, WSP, AC0, AC2, and AC3 remain unchanged.

No I/O Transfer

NIO *[f] device*



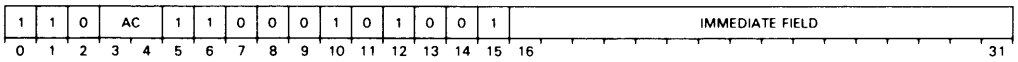
Used when a BUSY or DONE flag must be changed with no other operation taking place.

Sets the BUSY and DONE flags in the specified device according to the function specified by *F*.

NOTE: *The NIO[f] CPU instructions are reserved or assigned a function. For instance, the NIOS CPU (INTEN) is the interrupt enable instruction.*

Narrow Load Immediate

NLDAI *i,ac*

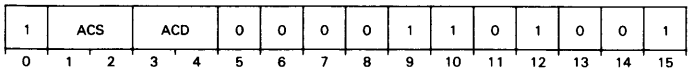


Loads an accumulator with the sign extended contents of an immediate value.

Sign extends the 16-bit, two's complement literal value contained in the immediate field to 32 bits. Loads the result of the sign extension into the specified accumulator. Carry is unchanged and *overflow* is 0.

Narrow Multiply

NMUL *acs,acd*

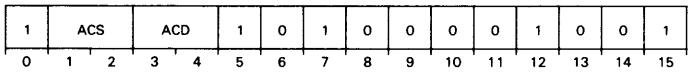


Multiplies an integer in an accumulator by an integer in another accumulator.

Multiplies the signed integer contained in bits 16–31 of ACD by the signed integer contained in bits 16–31 of ACS. If the result is outside the range of −32,768 to +32,767 inclusive, sets *overflow* to 1; otherwise, *overflow* is 0. Sign extends the lower 16 bits of the result to 32 bits and places these 32 bits in ACD. The contents of ACS and carry remain unchanged.

Narrow Negate

NNEG *acs,acd*



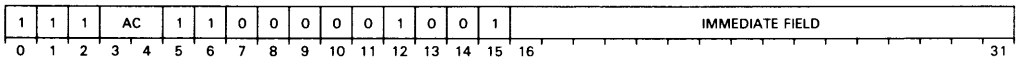
Negates an integer contained in an accumulator.

Negates the 16 least significant bits of ACS by performing a two's complement subtract from zero. Sign extends these 16 bits to 32 bits and loads the result in ACD. Sets carry to the value of ALU carry.

NOTE: *Negating the largest negative 16-bit integer (100000_8) sets overflow to 1.*

Narrow Skip on All Bits Set in Accumulator

NSALA *i,ac*

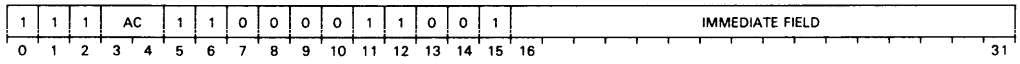


Logically ANDs the value in the immediate field with the complement of the contents of an accumulator and skips depending on the result of the AND.

Performs a logical AND on the contents of the immediate field and the complement of the least significant 16 bits contained in the specified accumulator. If the result of the AND is zero, then the next sequential word is skipped. If the result of the AND is nonzero, the next sequential word is executed. The contents of the specified accumulator remain unchanged. Carry is unchanged and *overflow* is 0.

Narrow Skip on All Bits Set in Memory Location

NSALM *i,ac*

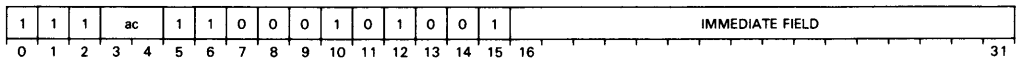


Logically ANDs the contents of an immediate field with the complement of the contents of a memory word and skips depending on the result.

Performs a logical AND on the contents of the immediate field and the complement of the word addressed by the specified accumulator. If the result of the AND is zero, then the next sequential word is skipped. If the result of the AND is nonzero, then the next sequential word is executed. The contents of the specified accumulator and memory location remain unchanged. Carry is unchanged and *overflow* is 0.

Narrow Skip on Any Bit Set in Accumulator

NSANA *i,ac*

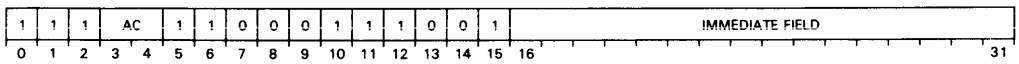


Logically ANDS the contents of an immediate field with the contents of an accumulator and skips depending on the result.

Performs a logical AND on the contents of the immediate field and the least significant 16 bits contained in the specified accumulator. If the result of the AND is nonzero, the next sequential word is skipped. If the result of the AND is zero, the next sequential word is executed. The contents of the specified accumulator remain unchanged. Carry is unchanged and *overflow* is 0.

Narrow Skip on Any Bit Set in Memory Location

NSANM *i,ac*

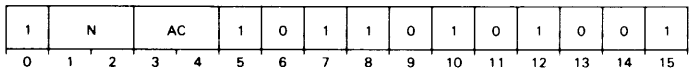


Logically ANDs the contents of an immediate field with the contents of a memory word and skips depending on the result.

Performs a logical AND on the contents of the immediate field and the contents of the word addressed by the specified accumulator. If the result of the AND is nonzero, then the next sequential word is skipped. If the result of the AND is zero, the next sequential word is executed. The contents of the specified accumulator and memory location remain unchanged. Carry is unchanged and *overflow* is 0.

Narrow Subtract Immediate

NSBI *n,ac*



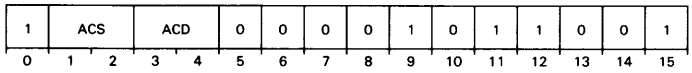
Subtracts a value in the range of 1 to 4 from the value contained in an accumulator.

The instruction subtracts the value $n+1$ from the 16-bit value contained in the specified accumulator, where *n* is an integer in the range of 0 to 3. Stores the result in bits 16–31 of the specified accumulator. Sign extends the specified accumulator to 32 bits. Sets carry to the value of ALU carry. Sets *overflow* to 1 if there is an ALU overflow.

> **NOTE:** *The assembler takes the coded value of* n *and subtracts 1 from it before placing it in the immediate field. Therefore, the programmer should code the exact value to be subtracted.*

Narrow Subtract

NSUB *acs,acd*

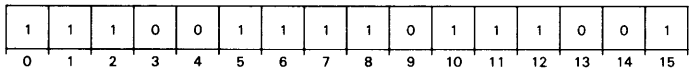


Subtracts an integer in an accumulator from an integer in another accumulator.

Subtracts the 16-bit integer contained in bits 16–31 of ACS from the 16-bit integer contained in bits 16–31 of ACD. Stores the result in bits 16–31 of ACD. Sign extends ACD to 32 bits. Sets carry to the value of ALU carry and *overflow* to 1, if there is an ALU overflow.

OR Referenced Bits

ORFB



Performs an inclusive OR on the referenced bits and the contents of a word string.

AC1 contains a pageframe number in bits 13–31. Bits 28–31 of AC1 are set to 0 so that the initial page frame number is a multiple of 16.

AC0 contains an origin 0 pageframe count that specifies the number of groups of 16 referenced bits to reset. A count of 0 (or any negative count except $8000\ 0000_{16}$) means that the instruction resets 16 pageframes.

AC2 contains the starting address of a word string. The instruction will inclusively OR the contents of this word string with the referenced bits.

The instruction fetches the referenced bits of 16 consecutive pageframes, beginning with the pageframe specified by AC1. Inclusively ORs these 16 bits with the 16-bit word specified by AC2. Stores the result of the OR in the location specified by AC2. Resets the 16 referenced bits to 0, decrements AC0 by 1, increments AC1 by 16, and increments AC2 by 1.

If the contents of AC0 are 0, the instruction performs one final iteration through the loop and then terminates execution. If AC0 contains a positive value, the instruction

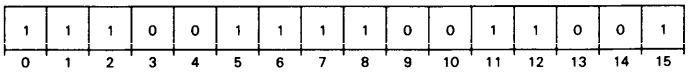
continues the ORing process with the next 16 referenced bits specified by AC1 and the word specified by AC2. Carry is unchanged and *overflow* is 0.

NOTE: *If AC1 contains a nonexistent pageframe number or if the address translator is not enabled when this instruction executes, the result of the instruction is undefined.*

This is a privileged instruction.

Purge the Address Translator

PATU



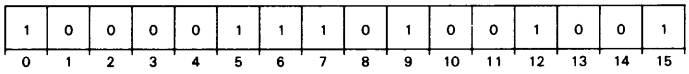
Purges the entire address translator of all entries. Carry is unchanged and *overflow* is 0.

NOTE: *The address translator need not be purged when page table entries (PTEs) are validated in the page tables, only when PTEs are either invalidated or changed.*

This is a privileged instruction.

Pop Block and Execute

PBX



The *Pop Block and Execute* (**PBX**) instruction, used in conjunction with the **BKPT** instruction, returns program control from the breakpoint handler.

The **PBX** instruction

1. Disables the interrupt system for one instruction execution;
2. Temporarily saves the one-word opcode in AC0 bits 16–31 and performs a modified wide pop block function **WPOPB**;
3. Temporarily replaces the **BKPT** instruction with the temporarily saved one-word opcode and continues normal program flow.

When the bits in AC0 16–31 contain the first word of a multi-word instruction, the processor locates the remainder of the multi-word instruction beginning at PC +1. (The PC references the **BKPT** instruction, effectively substituting the 16-bit instruction in AC0 for the **BKPT** instruction referenced by the PC after the pop.)

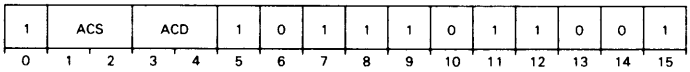
If the value popped off the stack and loaded into the PC does not reference the **BKPT** instruction, undefined results occur.

If an interrupt occurs while executing the saved instruction (PC points to the **BKPT** instruction), the processor sets the IXCT flag in the PSR and pushes the opcode of the saved instruction on the wide stack. Upon returning from the interrupt handler, the **BKPT** instruction tests the IXCT flag. If the flag is set, the **BKPT** instruction resets the flag to 0. Then it pops the saved opcode of the interrupted instruction off the wide stack and executes it.

Carry and *overflow* are indeterminate as a result of executing the **PBX** instruction. Executing the instruction in AC0 determines the value of the processor flags.

Program I/O

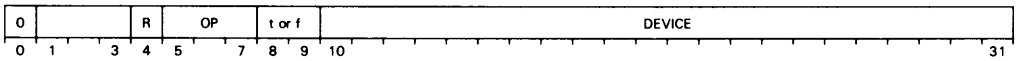
PIO *acs,acd*



Issues a programmed I/O command to an I/O device via the specified I/O channel.

Bits 16–31 of ACS contain the command.

The command to the I/O device must have the form:



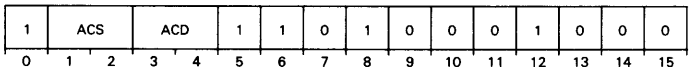
The instruction issues the command contained in ACS to the specified device. Bits 17–19 specify the I/O channel. Bits 21-23 (OP) contain the I/O operation code (bits 5-7 of an I/O instruction). Bits 8 and 9 (PULSE) contain the device flag command (bits 8 and 9 of an I/O instruction).

The instruction performs the specified operation, using bits 16–31 of ACD as the source or destination of the specified transfer. If ACD is to be the destination of data from the specified device, the transfer stores the data in bits 16–31 of ACD. Bits 0–15 of ACS are undefined. Carry is unchanged and *overflow* is 0.

> **NOTE:** *Execution of NIO or Skip instructions when ACD is other than AC0 produces results defined on an implementation basis only.*

Pop Multiple Accumulators

POP *acs,acd*



Pops one to four words off the stack and places them in the indicated accumulators.

The set of accumulators from ACS through ACD, bits 16–31, is filled with words popped from the stack. Bits 16–31 of the accumulators are filled in descending order, starting with bits 16–31 of the accumulator specified by ACS and continuing down through bits 16–31 of the accumulator specified by ACD, wrapping around if necessary, with AC3 following AC0. If ACS is equal to ACD, only one word is popped and it is placed in ACS.

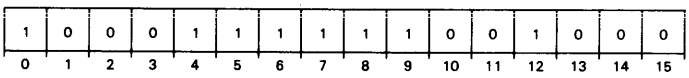
The stack pointer is decremented by the number of accumulators popped and the frame pointer is unchanged. A check for underflow is made only after the entire pop operation is done.

Bits 0–15 of the modified accumulator are undefined after completion of this instruction.

This instruction leaves carry unchanged; *overflow* is 0.

Pop Block

POPB



Returns control from an XOP0 Extended operation or an I/O interrupt handler that does not use the stack change facility of the *Vector* instruction (VCT).

Five words are popped off the stack and placed in predetermined locations. The words popped and their destinations are as follows:

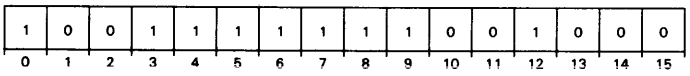
Word Popped	Destination
1	Bit 0 is loaded into carry
	Bits 1-15 are loaded into the PC
2	AC3
3	AC2
4	AC1
5	AC0

A check for underflow is made after the entire pop operation finishes. Sequential operation is continued with the word addressed by the updated value of the program counter. Carry remains unchanged and *overflow* is 0.

Bits 0–15 of the modified accumulator are undefined after completion of this instruction.

Pop PC and Jump

POPJ



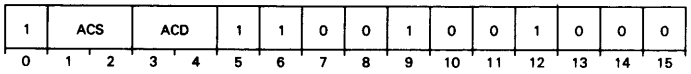
Pops the top word off the stack and places it in the program counter. Sequential operation continues with the word addressed by the updated value of the program counter.

The 31-bit effective address generated by this instruction is constrained to be within the first 64 Kbytes of the current segment.

The stack pointer is decremented by one and the frame pointer is unchanged. A check for underflow occurs after the pop operation. Carry remains unchanged; *overflow* is 0.

Push Multiple Accumulators

PSH *acs,acd*



Pushes the contents of one to four accumulators onto the narrow stack.

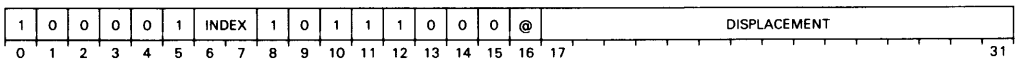
Bits 16–31 of the set of accumulators from ACS through ACD are pushed onto the stack. The contents of bits 16–31 of the accumulators are pushed in ascending order,

starting with bits 16–31 of the AC specified by ACS and continuing up through bits 16–31 of the AC specified by ACD, wrapping around if necessary, with AC0 following AC3. The contents of the accumulators remain unchanged. If ACS equals ACD, only ACS is pushed. Carry remains unchanged and *overflow* is 0.

The stack pointer is incremented by the number of accumulators pushed and the frame pointer is unchanged. A check for overflow is made only after the entire push operation finishes.

Push Jump

PSHJ *[@]displacement[,index]*



Pushes the address of the next sequential instruction onto the narrow stack and loads the program counter with an effective address.

Pushes the address of the next sequential instruction onto the stack, computes the effective address, *E*, and places it in the program counter. Sequential operation continues with the word addressed by the updated value of the program counter.

The 31-bit effective address generated by this instruction is constrained to be within the first 64 Kbytes of the current segment.

Carry remains unchanged and *overflow* is 0.

Push Return Address

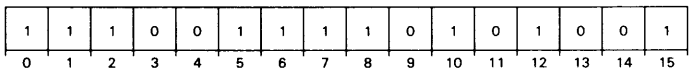
PSHR

Pushes the address of this instruction plus two onto the narrow stack.

A check for overflow is made after the push operation finishes. Carry remains unchanged and *overflow* is 0.

Reset Referenced Bits

RRFB



Resets the specified referenced bits.

AC1 contains a pageframe number in bits 13–31. Bits 28–31 are cleared to 0 so that the initial pageframe number is a multiple of 16.

AC0 contains an origin 0 pageframe count that specifies one less than the number of groups of 16 referenced bits to reset. A count of 0 (or any negative count except 8000 0000_{16}) means that the instruction resets 16 pageframes.

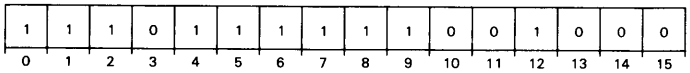
The instruction sets to 0 the referenced bits of 16 contiguous pageframes, starting with the pageframe specified by the contents of AC1. Decrements the contents of AC0 by 1 and increments the contents of AC1 by 16.

If AC0 contains a non-negative number after the decrement, the instruction repeats the operation with the next 16 pageframes. If AC0 contains a negative number, the instruction ends. Carry is unchanged and *overflow* is 0.

NOTE: *If AC0 specifies a nonexistent pageframe or if the address translator is not enabled when this instruction executes, the result of the instruction is undefined.*

This is a privileged instruction.

Restore
RSTR



Returns control from certain types of I/O interrupts.

Pops nine words off the stack and places them in predetermined locations. The words popped and their destinations are as follows:

Word Popped	Destination
1	Bit 0 is loaded into carry
	Bits 1-15 are loaded into the PC
2	AC3
3	AC2
4	AC1
5	AC0
6	Stack fault address
7	Stack limit
8	Frame pointer
9	Stack pointer

Sequential operation continues with the word addressed by the updated value of the program counter.

Bits 0–15 of the modified accumulators are undefined after completion of this instruction.

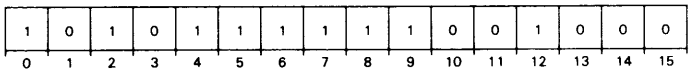
Carry remains unchanged and *overflow* is 0.

NOTES: *Use the* Restore *instruction to return control to the program only if the I/O interrupt handler uses the stack change facility (Mode E) of the* C/350 Vector on Interrupting Device Code *instruction (VCT).*

The Restore *instruction does not check for stack underflow.*

Return

RTN



Returns control from subroutines that issue a *Save* instruction at their entry points.

The *Save* instruction loads the current value of the stack pointer into the frame pointer. The *Return* instructions uses this value of the frame pointer to pop a standard return block off of the stack. The format of the return block is:

Word Popped	Destination
1	Bit 0 is loaded into carry
	Bits 1-15 are loaded into the PC
2	AC3
3	AC2
4	AC1
5	AC0

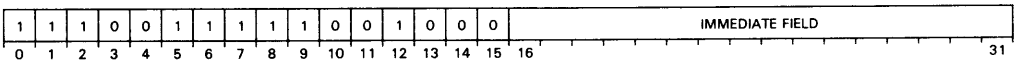
After popping the return block, the *Return* instruction loads the decremented value of the frame pointer into the stack pointer and the popped value of AC3 into the frame pointer.

Bits 0–15 of the modified accumulator are undefined after completion of this instruction.

Carry remains unchanged and *overflow* is 0.

Save

SAVE *i*



Saves information in the narrow stack that is required by the *Return* instruction.

Saves the current value of the stack pointer in a temporary location. Adds five plus the unsigned, 16-bit integer contained in the immediate field to the current value of the stack pointer and loads the result into the narrow stack pointer. Compares this new value of the stack pointer to the stack limit to check for overflow. If no overflow condition exists, then the instruction places the current value of the frame pointer in bits 16–31 of AC3. Fetches the contents of the temporary location and loads them into the frame pointer. The instruction uses the value in the frame pointer to push a five-word return block. The formats and contents of the five-word return block are as follows.

Word Pushed	Contents
1	Bits 16-31 of AC0
2	Bits 16-31 of AC1
3	Bits 16-31 of AC2
4	Frame pointer before the *Save*
5	Bit 0 = carry
	Bits 1-15 = bits 16-31 of AC3

After pushing the return block on the narrow stack, the instruction places the value of the frame pointer (which now contains the old value of the stack pointer plus five) in bits 16–31 of AC3. Carry remains unchanged and *overflow* is 0.

If an overflow condition exists, the *Save* instruction transfers control to the stack fault routine. The program counter in the fault return block contains the address of the *Save* instruction.

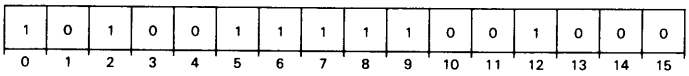
The *Save* instruction allocates a portion of the stack for use by the procedure which executed the *Save*. The value of the *frame size*, contained in the immediate field, determines the number of words in this stack area. This portion of the stack will not normally be accessed by push and pop operations, but will be used by the procedure for temporary storage of variables, counters, etc. The frame pointer acts as the reference point for this storage area.

The 31-bit effective address generated by this instruction is constrained to be within the first 64 Kbytes of the current segment.

Use the *Save* instruction with the *Jump to Subroutine* instruction. The *Jump to Subroutine* instruction places the return value of the program counter in bits 16–31 of AC3. *Save* then pushes the return value (contents of bits 16–31 of AC3) into bits 1–15 of the fifth word pushed.

Save Without Arguments

SAVZ



Saves the information required by the *Return* instruction.

Saves the current value of the stack pointer in a temporary location. Adds 5 to the current value of the stack pointer and loads the result into narrow stack pointer. Compares this new value of the stack pointer to the stack limit to check for overflow. If no overflow condition exists, then the instruciton places the current value of the frame pointer in bits 16—31 of AC3. Fetches the contents of the temporary location and loads them into the frame pointer. The instruction uses the value in the frame pointer to push a 5-word return block. The formats and contents of the 5-word return block are as follows:

Word Pushed	Contents
1	Bits 16-31 of AC0
2	Bits 16-31 of AC1
3	Bits 16-31 of AC2
4	Frame pointer before the *Save Without Arguments*
5	Bit 0 = carry
	Bits 1-15 = bits 16-31 of AC3

After pushing the return block on the narrow stack, the instruction places the value of the frame pointer (which now contains the old value of the stack pointer plus five) in bits 16–31 of AC3. Carry remains unchanged and *overflow* is 0.

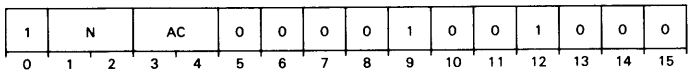
If an overflow condition exists, the *Save Without Arguments* instruction transfers control to the stack fault routine. The program counter in the fault return block contains the address of the *Save Without Arguments* instruction.

The 31-bit effective address generated by this instruction is constrained to be within the first 64 Kbytes of the current segment.

Use the *Save Without Arguments* instruction with the *Jump to Subroutine* instruction. The *Jump to Subroutine* instruction places the return value of the program counter in bits 16–31 of AC3. *Save Without Arguments* then pushes the return value (contents of bits 16–31 of AC3) into bits 1–15 of the fifth word pushed.

Subtract Immediate

SBI *n,ac*



Subtracts an unsigned integer in the range 1 to 4 from the contents of an accumulator.

The instruction subtracts the value $n+1$ from the unsigned 16-bit number contained in bits 16–31 of the specified accumulator and the result is placed in bits 16–31 of AC. Carry remains unchanged. *Overflow* is 0.

Bits 0–15 of the modified accumulator are undefined after completion of this instruction.

NOTE: *The assembler takes the coded value of* n *and subtracts 1 from it before placing it in the immediate field. Therefore, the programmer should code the exact value to be subtracted.*

Example

Assume that bits 16–31 of AC2 contain 000003_8. After the instruction **SBI 4,2** is executed, bits 16–31 of AC2 contain 177777_8 and carry remains unchanged (see Figure 10.8.)

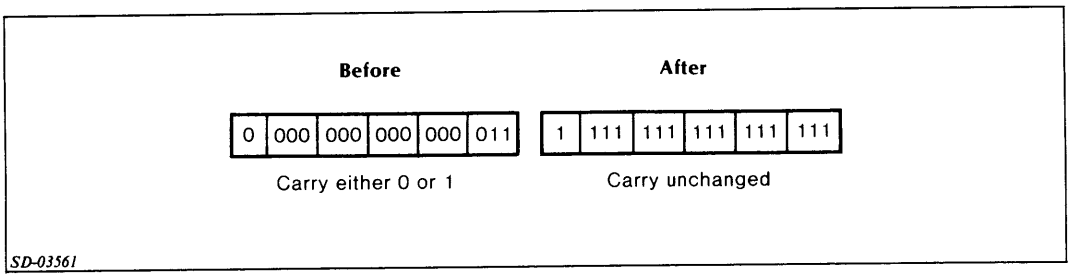
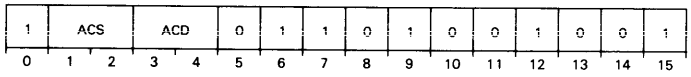


Figure 10.8 SBI example

Sign Extend

SEX *acs,acd*

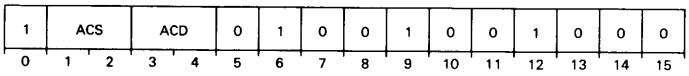


Sign extends the 16-bit integer in an accumulator to 32 bits.

Sign extends the 16-bit integer contained in ACS to 32 bits and loads the result into ACD. The contents of ACS remain unchanged, unless ACS and ACD are specified to be the same accumulator. Carry is unchanged and *overflow* is 0.

Skip if ACS Greater than or Equal to ACD

SGE *acs,acd*



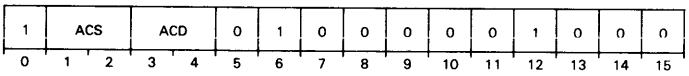
Compares two signed integers in two accumulators and skips if the first is greater than or equal to the second.

Algebraically compares the signed two's complement numbers in bits 16–31 of ACS and ACD. If the number in bits 16–31 of ACS is greater than or equal to the number in bits 16–31 of ACD, the next sequential word is skipped. The contents of ACS, ACD, and carry remain unchanged. *Overflow* is 0.

NOTE: *The* Skip if ACS Greater than ACD *and* Skip if ACS Greater than or Equal to ACD *instructions treat the contents of the specified accumulators as signed, two's complement integers. To compare unsigned integers, use the* Subtract and Add Complement *instruction.*

Skip if ACS Greater than ACD

SGT *acs,acd*



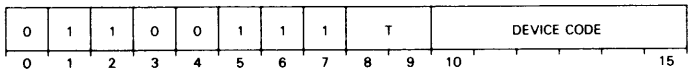
Compares two signed integers in two accumulators and skips if the first is greater than the second.

Algebraically compares the signed, two's complement numbers in bits 16–31 of ACS and ACD. If the number in bits 16–31 of ACS is greater than the number in bits 16–31 of ACD, the next sequential word is skipped. The contents of ACS, ACD, and carry remain unchanged. *Overflow* is 0.

NOTE: *The* Skip if ACS Greater than ACD *and* Skip if ACS Greater than or Equal to ACD *instructions treat the contents of the specified accumulators as signed, two's complement integers. To compare unsigned integers, use the* Subtract and Add Complement *instruction.*

I/O Skip

SKP*t* *device*



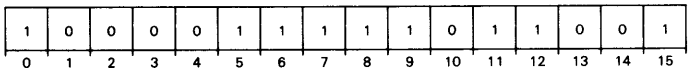
The SKP instruction tests the BUSY and DONE status flags of devices. If the test condition specified by *t* is true, the instruction skips the next sequential word.

The following list gives the test *t* conditions.

Symbol *t*	Value	Test
BN	0 0	Skip on BUSY = 1
BZ	0 1	Skip on BUSY = 0
DN	1 0	Skip on DONE = 1
DZ	1 1	Skip on DONE = 0

Store Modified and Referenced Bits

SMRF



Stores new values into the modified and referenced bits of a pageframe.

AC1 contains a pageframe number in bits 13–31.

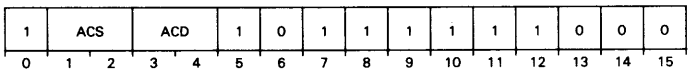
The instruction fetches the contents of the two least significant bits of AC0. Stores these values in the modified (bit 30) and referenced (bit 31) bits of the pageframe specified by AC1. Carry is unchanged and *overflow* is 0.

If the address translator is not enabled, undefined results will occur. If a nonexistent pageframe is specified, the results are indeterminate.

NOTE: *This is a privileged instruction.*

Skip on Nonzero Bit

SNB *acs,acd*



The two accumulators form a bit pointer. If the addressed bit is 1, the next sequential word is skipped.

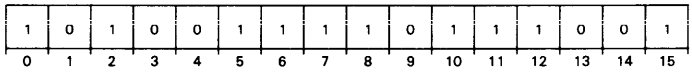
Forms a 32-bit bit pointer from the contents of bits 16–31 of both ACS and ACD. Bits 16–31 of ACS contain the high-order 16 bits and bits 16–31 of ACD contain the low-order 16 bits of the bit pointer. If ACS and ACD are specified as the same accumulator, the instruction treats the accumulator contents as the low-order 16 bits of the bit pointer and assumes the high-order 16 bits are 0.

If the addressed bit in memory is 1, the next sequential word is skipped. The contents of ACS, ACD, and carry remain unchanged. *Overflow* is 0.

The 31-bit effective address generated by this instruction is constrained to be within the first 64 Kbytes of the current segment.

Skip on OVR Reset

SNOVR



Tests the value of *OVR*. If the flag has the value 0, the next sequential word is skipped. If the flag has the value 1, the next sequential word is executed. Carry is unchanged and *overflow* is 0.

Store Processor Status Register From AC0

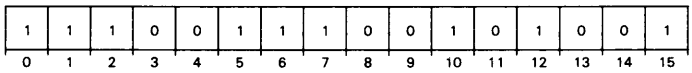
SPSR

Stores the most significant bits of AC0 in the defined corresponding bits of PSR.

The contents of AC0 remain unchanged. Carry is unchanged and *overflow* is 0.

Store Page Table Entry

SPTE



Writes Page Table Entry (PTE) data to a physical address.

AC0 contains the new PTE data. AC1 contains the logical address which will use the new PTE data. AC2 contains the physical address of the PTE slot to update.

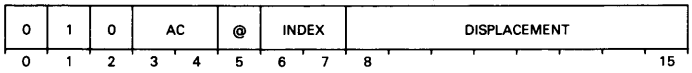
NOTE: *You should make sure that the specified logical address in AC1 will produce the physical address in AC2 of the last PTE accessed.*

The SPTE instruction writes the data in AC0 to the physical address specified in AC2 and does any updates to the hardware address translation mechanism that are necessary for consistency (the logical address in AC1 is required for this operation). Refer to the functional characteristics manual for the effects this instruction produces on your machine's address translation cache.

NOTE: *This is a privileged instruction.*

Store Accumulator

STA *ac,[@]displacement[,index]*



Stores the contents of bits 16–31 of an accumulator into a memory location.

Places the contents of bits 16-31 of the specified accumulator in the word addressed by the effective address, *E*. The previous contents of the location addressed by *E* are lost.

The contents of carry and the specified accumulator remain unchanged. *Overflow* is 0.

The 31-bit effective address generated by this instruction is constrained to be within the first 64 Kbytes of the current segment.

Store Accumulator in WFP

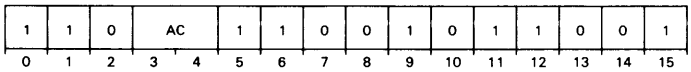
STAFP *ac*

Stores the contents of an accumulator in WFP.

Stores a copy of the contents of the specified accumulator into WFP (the wide frame pointer). Carry is unchanged and *overflow* is 0.

Store Accumulator in WSB

STASB *ac*

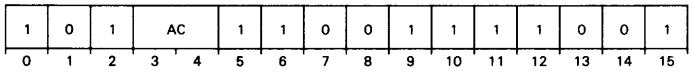


Stores the contents of an accumulator in WSB and updates the stack parameter in page zero of memory.

Stores a copy of the contents of the specified accumulator into WSB (the wide stack base). Carry is unchanged and *overflow* is 0.

Store Accumulator in WSL

STASL *ac*

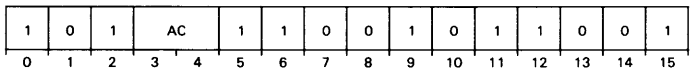


Stores the contents of an accumulator in WSL and updates the stack parameter in page zero of memory.

Stores a copy of the contents of the specified accumulator into WSL (the wide stack limit). Carry is unchanged and *overflow* is 0.

Store Accumulator in WSP

STASP *ac*

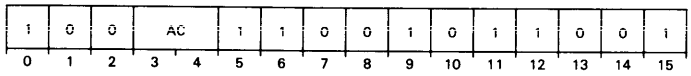


Stores the contents of an accumulator in WSP.

Stores a copy of the contents of the specified accumulator into WSP (the wide stack pointer). Carry is unchanged and *overflow* is 0.

Store Accumulator into Stack Pointer Contents

STATS *ac*

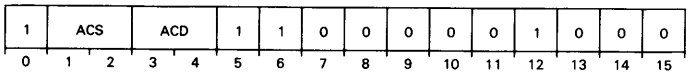


Stores the contents of an accumulator in the location addressed by WSP.

Uses the contents of WSP (the wide stack pointer) as the address of a double word. Stores a copy of the contents of the specified accumulator at the address contained in WSP. Carry is unchanged and *overflow* is 0.

Store Byte

STB *acs,acd*



Moves the rightmost byte of ACD to a byte in memory. ACS contains the byte pointer.

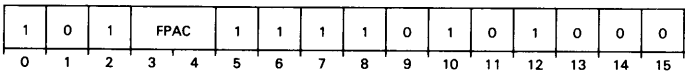
Places bits 24–31 of ACD in the byte addressed by the byte pointer contained in bits 16–31 of ACS.

The 31-bit effective address generated by this instruction is constrained to be within the first 64 Kbytes of the current segment.

The contents of ACS, ACD, and carry remain unchanged. *Overflow* is 0.

Store Integer

STI *fpac*



Converts the contents of a floating-point accumulator to a specified format and stores it in memory.

Under the control of accumulators AC1 and AC3, translates the contents of the specified FPAC to an integer of the specified type and stores it, right-justified, in memory, beginning at the specified location. The instruction leaves the floating-point number unchanged in the FPAC and destroys the previous contents of memory at the specified location(s). It also sets CARRY to 0.

Bits 16–31 of AC1 must contain the data-type indicator describing the integer.

Bits 16–31 of AC3 must contain a byte pointer which is the address of the high-order byte of the number in memory.

Upon successful completion, the instruction leaves accumulators AC0 and AC1 unchanged. AC2 contains the original contents of AC3 and AC3 contains a byte pointer which is the address of the next byte after the destination field. *Overflow* is 0.

The 31-bit effective address generated by this instruction is constrained to be within the first 64 Kbytes of the current segment.

NOTES: *If the number in the specified FPAC has any fractional part, the result of the instruction is undefined. Use the* Integerize *instruction to clear any fractional part.*

For data types 0 through 6, if the destination field cannot contain the entire number being stored, high-order digits are discarded until the number will fit into the destination. The remaining low-order digits are stored and carry is set to 1. For data type 7, the low-order digits are discarded.

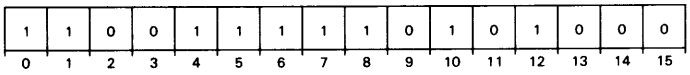
For data types 0, 1, 2, 3, 4, and 5, if the number being stored will not fill the destination field, the high-order bytes to the right of the sign are set to 0.

For data type 6, if the number being stored will not fill the destination field, the sign bit is extended to the left to fill the field.

For data type 7, if the number being stored will not fill the destination field, the low-order bytes are set to 0.

Store Integer Extended

STIX



Converts the contents of the four FPACs to an integer of data type 0,1,2,3,4, or 5 and uses the low-order eight digits of each to form a 32-digit integer.

The instruction stores this integer, right-justified, in memory beginning at the specified location. The sign of the integer is the logical OR of the signs of all four FPACs. The previous contents of the addressed memory locations are lost. Sets carry to 0. The contents of the FPACs remain unchanged. The condition codes in the FPSR are unpredictable.

Bits 16–31 of AC1 must contain the data-type indicator describing the form of the integer in memory.

Bits 16–31 of AC3 must contain a byte pointer which is the address of the high-order byte of the destination field in memory.

Upon successful termination, the contents of AC0 and AC1 remain unchanged; AC2 contains the original contents of AC3; and AC3 contains a byte pointer which is the address of the next byte after the destination field. *Overflow* is 0.

The 31-bit effective address generated by this instruction is constrained to be within the first 64 Kbytes of the current segment.

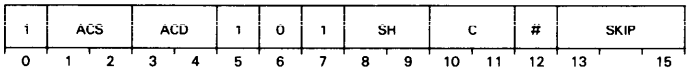
NOTES: *If the number in the specified FPAC has any fractional part, the result of the instruction is undefined. Use the* Intergerize *instruction to clear any fractional part.*

If the destination field is not large enough to contain the number being stored, the instruction disregards high-order digits until the number will fit in the destination. The instruction stores low-order digits remaining and sets carry to 1.

If the number being stored will not fill the destination field, the instruction sets the high-order bytes to 0.

Subtract

SUB*[c][sh][#] acs,acd[,skip]*



Performs unsigned integer subtraction and complements carry if appropriate.

Initializes carry to its specified value. The instruction subtracts the unsigned, 16-bit number in bits 16–31 of ACS from the unsigned, 16-bit number in bits 16–31 of ACD by taking the two's complement of the number in ACS and adding it to the number in ACD. The instruction places the result of the addition in the shifter. If the operation produces a result that is greater than 32,768, the instruction complements carry. The instruction performs the specified shift operation and places the result of the shift in bits 16–31 of ACD if the no-load bit is 0. If the skip condition is true, the instruction skips the next sequential word.

If the load option is specified, bits 0–15 of ACD are undefined.

Overflow is 0 for this instruction.

> **NOTE:** *If the number in ACS is less than or equal to the number in ACD, the instruction complements carry.*

[c]

The processor determines the effect of the CARRY flag (c) on the old value of CARRY before performing the operation (opcode). The following list gives the values of c, bits 10 and 11, and the operation.

Symbol *[c]*	Bits 10-11	Operation
omitted	0 0	Leave CARRY unchanged
Z	0 1	Initialize CARRY to 0
O	1 0	Initialize CARRY to 1
C	1 1	Complement CARRY

[sh]

The processor shifts the CARRY flag and the 16 data bits after performing the instruction operation. The processor can shift the bits left or right one bit position, or it can swap the two bytes. The following list gives the values of sh, bits 8 and 9, and the shift operation.

Symbol *[sh]*	Bits 8-9	Shift Operation
omitted	0 0	Do not shift the result
L	0 1	Shift left
R	1 0	Shift right
S	1 1	Swap the two 8-bit bytes

[#]

Unless you use the no-load option (#), the processor loads the result of the shift operation into the destination accumulator. The no-load option is useful to test the result of the instruction operation without destroying the destination accumulator contents. The following list gives the values of the no-load option, bit 12, and the operation.

Symbol *[#]*	Bit 12	Operation
omitted	0	Load the result into ACD
#	1	Do not load the result and restore the CARRY flag

NOTE: *Do not specify an instruction with the no-load option (#) in combination with either the never skip or always skip option. Thus, the instruction may not end in 1000_2 or 1001_2, other instructions use the bit combinations.*

[skip]

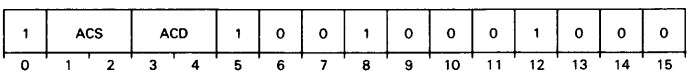
The processor can skip the next instruction if the condition test is true. The following list gives the test conditions, bits 13 to 15, and the operation.

Symbol *[skip]*	Bits 13-15	Operation
omitted	0 0 0	Never skip
SKP	0 0 1	Always skip
SZC	0 1 0	Skip if CARRY is 0
SNC	0 1 1	Skip if CARRY is not 0
SZR	1 0 0	Skip if the result is 0
SNR	1 0 1	Skip if the result is not 0
SEZ	1 1 0	Skip if either CARRY or the result is 0
SBN	1 1 1	Skip if both CARRY and the result are not 0

When the instruction performs a skip, it skips the next sequential 16-bit word. Make sure that a skip does not transfer control to the middle of a 32-bit or longer instruction.

Skip on Zero Bit

SZB *acs,acd*



The two accumulators form a bit pointer. If the addressed bit is 0, the next sequential word is skipped.

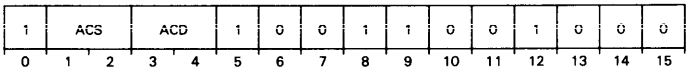
Forms a 32-bit bit pointer from the contents of bits 16–31 of both ACS and ACD. Bits 16–31 of ACS contain the high-order 16 bits and bits 16–31 of ACD contain the low-order 16 bits of the bit pointer. If ACS and ACD are specified as the same accumulator, the instruction treats the accumulator contents as the low-order 16 bits of the bit pointer and assumes the high-order 16 bits are 0.

If the addressed bit in memory is 0, the next sequential word is skipped. The contents of ACS, ACD, and carry remain unchanged. *Overflow* is 0.

The 31-bit effective address generated by this instruction is constrained to be within the first 64 Kbytes of the current segment.

Skip on Zero Bit and Set to One

SZBO *acs,acd*



The two accumulators form a bit pointer. The instruction sets the addressed bit to 1. If the addressed bit was 0 before being set to 1, the instruction skips the next sequential word.

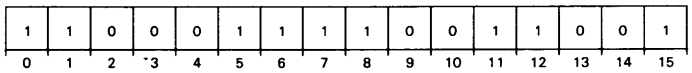
Forms a 32-bit bit pointer from the contents of bits 16–31 of ACS and ACD. Bits 16–31 of ACS contain the high-order 16 bits and bits 16–31 of ACD contain the low-order 16 bits of the bit pointer. If ACS and ACD are specified as the same accumulator, the instruction treats the accumulator contents as the low-order 16 bits of the bit pointer and assumes the high-order 16 bits are 0.

The contents of ACS, ACD, and carry remain unchanged. *Overflow* is 0. The 31-bit effective address generated by this instruction is constrained to be within the first 64 Kbytes of the current segment.

NOTE: *This instruction facilitates the use of bit maps for such purposes as allocation of facilities (memory blocks, I/O devices, etc.) to several processes, or tasks, that may interrupt one another, or in a multiprocessor environment. The bit is tested and set to 1 in one memory cycle.*

Skip on Valid Byte Pointer

VBP



Checks a byte pointer for a valid ring-structured reference and if valid it skips the next word.

NOTE: *An invalid access (read, write, or execute) generates a protection violation.*

AC0 contains a 32-bit byte pointer. AC1 contains a segment number in bits 1–3; all other bits must contain zeroes.

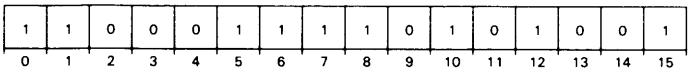
The instruction, executing in a lower segment, compares the segment number in AC0 to the segment number in AC1 and to the current segment. The byte pointer is valid and the next sequential instruction is skipped if the segment number in AC0 is equal to or greater than the segment number in AC1 and is equal to the current segment.

Otherwise, the byte pointer is invalid and the next sequential word is executed.

Carry is unchanged and *overflow* is 0.

Skip on Valid Word Pointer

VWP



Checks a word pointer for a valid ring-structured reference and if valid it skips the next word.

> **NOTE:** *An invalid access (read, write, or execute) or more than 15 indirect addresses generates a protection violation.*

AC0 contains a 31-bit, indirectable word pointer. AC1 contains a segment number in bits 1–3; all other bits must contain zeroes.

The instruction, executing in a lower segment, compares the segment number in AC0 to the segment number in AC1 and then to the current segment. The word pointer is valid and the next sequential instruction is skipped if all of the following conditions are true:

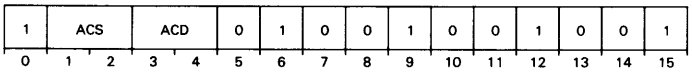
1. The segment number in AC0 is equal to or greater than the segment number in AC1.
2. The segment number in an indirect address is equal to or greater than the segment number in AC1 and the currently referenced segment.
3. If an indirection to a higher numbered segment is followed by another indirection, the subsequent indirection(s) must be to the same (or higher) segment.
4. The segment number in the effective address (specified by AC0) is equal to or greater than the segment number in AC1 and the current segment.

Otherwise, the word pointer is invalid and the next sequential word is executed.

Carry is unchanged and *overflow* is 0.

Wide Add Complement

WADC *acs,acd*



Adds the logical complement of the 32-bit integer specified by ACS to the 32-bit integer specified by ACD.

Forms the logical complement of the 32-bit integer contained in ACS and adds it to the 32-bit integer contained in ACD. Stores the result in ACD. Sets carry to the value of ALU carry. Sets *overflow* to 1 if there is an ALU overflow.

Wide Add

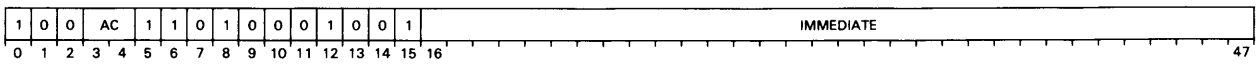
WADD *acs,acd*

Adds the 32-bit integers specified by two accumulators.

Adds the 32-bit fixed-point integer contained in ACS to the 32-bit fixed-point integer contained in ACD. Stores the result in ACD. Sets carry to ALU carry. Sets *overflow* to 1, if there is an ALU overflow.

Wide Add With Wide Immediate

WADDI *i,ac*

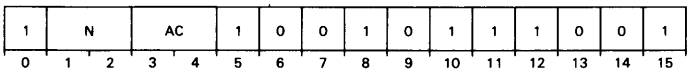


Adds the 32-bit integer in the immediate field to the 32-bit integer in the specified accumulator.

Adds the 32-bit fixed-point integer contained in the immediate field to the 32-bit fixed-point integer contained in the specified accumulator. Stores the result in the specified accumulator. Sets *overflow* to 1 if there is an ALU overflow. Sets carry to the value of the ALU carry.

Wide Add Immediate

WADI *n,ac*



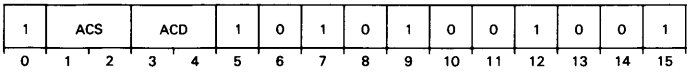
Adds the value specified by $n+1$ to the 32-bit integer in the specified accumulator.

Adds the value $n+1$ to the 32-bit fixed-point integer contained in the specified accumulator. Stores the result in the specified accumulator. Sets carry to the value of ALU carry. Sets *overflow* to 1, if there is an ALU overflow.

> **NOTE:** *The assembler takes the coded value of* n *and subtracts one from it before placing it in the immediate field. Therefore, the programmer should code the exact value to be added.*

Wide AND with Complemented Source

WANC *acs,acd*

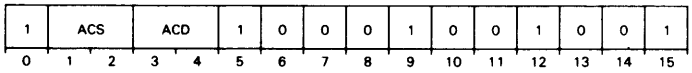


Logically ANDs the one's complement of the 32 bits specified by ACS with the 32 bits specified by ACD.

Forms the one's complement of the 32 bits contained in ACS and logically ANDs it with the 32 bits contained in ACD. Stores the result in ACD. Carry is unchanged and *overflow* is 0.

Wide AND

WAND *acs,acd*

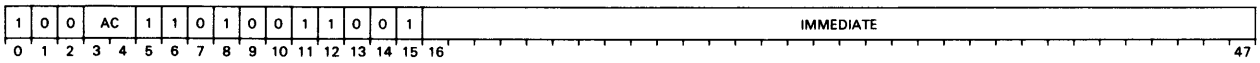


Logically ANDs the contents of ACS and ACD.

Forms the logical AND between corresponding bits of ACS and ACD. Loads the 32-bit result into ACD. The contents of ACS remain unchanged. Carry is unchanged and *overflow* is 0.

Wide AND Immediate

WANDI *i,ac*

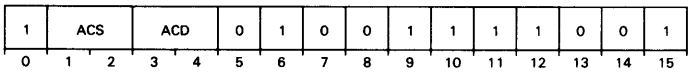


Logically ANDs the contents of the specified accumulator with the contents of the immediate field.

Forms the logical AND between corresponding bits of the specified accumulator and the value contained in the literal field. The instruction places the 32-bit result of the logical AND in the specified accumulator. Carry is unchanged and *overflow* is 0.

Wide Arithmetic Shift

WASH *acs,acd*



Shifts the contents of ACD left or right.

Bits 24–31 of ACS specify the number of bits to shift and the direction of shifting.

If ACS contains a positive number, the instruction shifts the contents of ACD left; zeroes fill the vacated bit positions. If ACS contains a negative number, the instruction shifts the contents of ACD right; the sign bit fills the vacated bit positions. If ACS contains zero, no shifting occurs. The instruction ignores bits 0–23 of ACS.

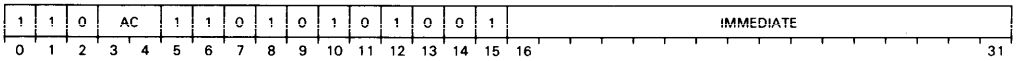
If the instruction is to shift the contents of ACD to the right, it truncates the contents one bit position for each shift.

In shifting negative numbers to the right, rounding towards zero is performed. For instance, −3 shifted one position to the right results in −1.

The value of ACS and carry remain unchanged. If, while performing a left shift, you shift out a bit whose value is the complement of ACD's sign bit, the result is correct, but *overflow* is set to 1. Otherwise, *overflow* is 0.

Wide Arithmetic Shift With Narrow Immediate

WASHI *i,ac*



Shifts the contents of an accumulator left or right.

Bits 24–31 of the immediate field specify the number of bits to shift and the direction of shifting. If the immediate contains a positive number (1 to 32_{10}), the instruction shifts the contents of AC left; zeroes fill the vacated bit positions. If the immediate field contains a negative number (-1 to -32_{10}), the instruction shifts the contents of AC right; the sign bit fills the vacated bit positions. If the immediate field contains zero, then no shifting occurs. Bits 16 to 23 of the immediate field must be identical to bit 24; otherwise, results are indeterminate. The processor sign extends the narrow immediate to 32 bits.

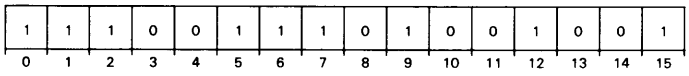
If the instruction is to shift the contents of AC to the right, it truncates the contents one bit position for each shift.

In shifting negative numbers to the right, rounding towards zero is performed. For instance, -3 shifted one position to the right results in -1.

The value of carry remains unchanged. If, while performing a left shift, you shift out a bit whose value is the complement of AC's sign bit, the result is correct, but *overflow* is set to 1; otherwise, *overflow* is 0.

Wide Block Move

WBLM



Moves words sequentially from one memory location to another, treating them as unsigned, 16-bit integers.

AC1 contains the two's complement of the number of words to be moved. If the contents of AC1 are positive, then data movement progresses from the lowest memory location to the highest (ascending). If the contents of AC1 are negative, then data movement progresses from the highest memory location to the lowest (descending).

Bits 1–31 of AC2 contain the address of the source location. Bits 1–31 of AC3 contain the address of the destination location. The address in bits 1–31 of AC2 or AC3 is an indirect address if bit 0 of that accumulator is 1. In that case, the instruction follows the indirection chain before placing the resultant effective address in the accumulator.

AC	Contents
0	Unused
1	Number of words to be moved
2	Source address
3	Destination address

For each word moved, the instruction increments or decrements the contents of AC1, AC2, and AC3 by 1 depending on the contents of AC1. If data movement is ascending

(AC1 is positive), the instruction decrements AC1 by 1, and increments AC2 and AC3 (the source and destination addresses, respectively) by 1 for each word moved. If data movement is descending (AC1 is negative), the instruction increments AC1 by 1, and decrements AC2 and AC3 by 1 for each word moved.

Upon completion of the instruction, AC1 contains zeroes, and AC2 and AC3 point to the word following (ascending) or preceding (descending) the last word in their respective fields. AC0 is unused. Carry is unchanged and *overflow* is 0.

NOTES: *Since this instruction may require a long time to execute, it is interruptible. When this instruction is interrupted, the processor saves the address of the* **WBLM** *instruction. This instruction updates addresses and word count after storing each word, so any interrupt service routine returning control via the saved address will correctly restart the* **WBLM** *instruction.*

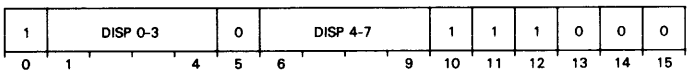
If data movement is descending and a ring crossing would occur, a protection trap occurs and this instruction does not execute. AC1 will contain the value 4.

If AC1 is zero, no words are moved.

When updating the source and destination addresses, the *Wide Block Move* instruction forces bit 0 of the result to 0. This ensures that upon return from an interrupt, the *Wide Block Move* instruction will not try to resolve an indirect address in either AC2 or AC3.

Wide Branch

WBR *displacement*



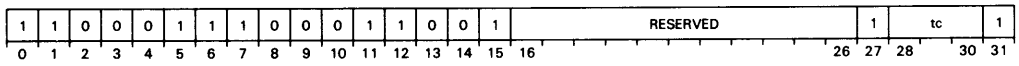
Adds a specified value to the program counter.

Adds the 31-bit value contained in the PC to the value of the displacement and places the result in the PC. Carry is unchanged and *overflow* is 0.

NOTE: *The processor always forces the value loaded into the PC to reference a location in the current segment of execution.*

Wide Backward Search Queue and Skip

WBStc



The WBStc instruction searches backward through a 32-bit data field. If the search is successful, it skips the next two single-word instructions.

AC1 and AC3 identify a location in a data field as a beginning data element in the queue search. AC1 contains an effective address. AC3 contains a double word offset (a signed integer). The processor locates the beginning data element by calculating the effective address and adding the word offset.

The double word at the top of the wide stack contains the mask word. The bits in the mask word idenfity the test location bits to sample.

Bits 28-30 of the WBStc instruction specify the search condition.

tc Value	Bits 28-30 Encoding *	Meaning
SS	0 0 0	Some of the sampled test location bits are 1
SC	0 0 1	Some of the sampled test location bits are 0
AS	0 1 0	All of the sampled test location bits are 1
AC	0 1 1	All of the sampled test location bits are 0
E	1 0 0	The mask and test location are equal
GE	1 0 1	The mask is greater than or equal to the test location
LE	1 1 0	The mask is less than or equal to the test location
NE	1 1 1	The mask and test location are not equal

*The instruction treats the values contained in the mask and in the test location as unsigned integers for the E, GE, LE, and NE test conditions.

The search begins with the addressed data element, and compares it with the 32-bit mask word. The search continues until the processor reaches either the head of the queue or the data element that meets the test condition.

If the search is successful, AC1 contains the effective address of the data element, and the processor then skips the next two single-word instructions. The processor will not honor interrupts between the time that it completes a successful search and executes the PC+4 instruction.

NOTE: *After a successful search terminates, a new beginning pointer must be placed in AC1 if the search is to be continued through the remainder of the queue.*

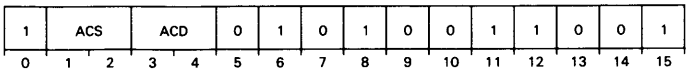
If the search fails, AC1 contains the effective address of the last data element searched, and the processor then executes the next instruction. The process will honor interrupts between the time that it completes an unsuccessful search and executes the PC+2 instruction.

If the processor interrupts the search, AC1 contains the effective address of the next data element to examine, and the processor then skips the next instruction. The processor will honor interrupts between the time that the interrupt occurs and the processor executes the PC+3 instruction.

For all returns, the contents of CARRY, OVR, WSP, AC0, AC2, and AC3 remain unchanged.

Wide Set Bit to One

WBTO *acs,acd*



Sets the specified bit to one.

ACS contains a 31-bit indirectable word address.

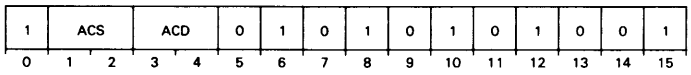
ACD contains a word offset and a bit identifier.

The contents of ACS and ACD remain unchanged. Carry is unchanged and *overflow* is 0.

If ACS and ACD are specified to be the same accumulator, then the processor assumes the word address is zero within the current segment. In this case, the specified accumulator contains a word offset and a bit identifier.

Wide Set Bit to Zero

WBTZ *acs,acd*



Sets the specified bit to zero.

ACS contains a 31-bit indirectable word address.

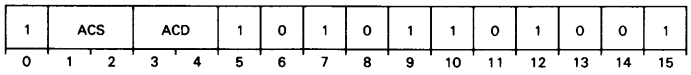
ACD contains a word offset and a bit identifier.

The contents of ACS and ACD remain unchanged. Carry is unchanged and *overflow* is 0.

If ACS and ACD are specified to be the same accumulator, then the processor assumes the word address is zero within the current segment. In this case, the specified accumulator contains a word offset and a bit identifier.

Wide Compare to Limits

WCLM *acs,acd*



Compares a signed integer with two limit values and skips if the integer is between the limit values. The accumulators determine the location of the limit values.

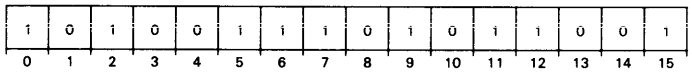
Compares the signed, two's complement integer in ACS to two signed, two's complement integer limit values, *L* and *H*. If the number in ACS is greater than or equal to *L* and less than or equal to *H*, execution skips the next sequential word before continuing. If the number in ACS is less than *L* or greater than *H*, execution continues with the next sequential word. Carry is unchanged and *overflow* is 0.

If ACS and ACD are specified as different accumulators, bits 1–31 of ACD contain the address of the double word that contains the limit value *L*. The double word following *L* contains the limit value *H*. Bit 0 of ACD is ignored.

If ACS and ACD are specified as the same accumulator, the integer to be compared must be in that accumulator and the limit values *L* and *H* must be in the two double words following the instruction. The first double word contains *L* and the second double word contains *H*. The fifth word contains the next instruction of the program.

Wide Character Compare

WCMP



Under control of the four accumulators, compares two strings of bytes and returns a code in AC1 reflecting the results of the comparison.

The instruction compares the strings one byte at a time. Each byte is treated as an unsigned 8-bit binary quantity in the range $0–255_{10}$. If two bytes are not equal, the string whose byte has the smaller numerical value is, by definition, the *lower valued* string. Both strings remain unchanged. The four accumulators contain parameters passed to the instruction. Two accumulators specify the starting address, the number of bytes, and the direction of processing (ascending or descending addresses) for each string.

AC0 specifies the length and direction of comparison for string 2. If the string is to be compared from its lowest memory location to the highest, AC0 contains the unsigned value of the number of bytes in string 2. If the string is to be compared from its highest memory location to the lowest, AC0 contains the two's complement of the number of bytes in string 2.

AC1 specifies the length and direction of comparison for string 1. If the string is to be compared from its lowest memory location to the highest, AC0 contains the unsigned value of the number of bytes in string 1. If the string is to be compared from its highest memory location to the lowest, AC1 contains the two's complement of the number of bytes in string 1.

AC2 contains a byte pointer to the first byte compared in string 2. When the string is compared in ascending order, AC2 points to the lowest byte. When the string is compared in descending order, AC2 points to the highest byte.

AC3 contains a byte pointer to the first byte compared in string 1. When the string is compared in ascending order, AC3 points to the lowest byte. When the string is compared in descending order, AC3 points to the highest byte.

Code	Comparison Result
−1	String 1 < String 2
0	String 1 = String 2
+1	String 1 > String 2

The strings may overlap in any way. Overlap will not affect the results of the comparison.

Upon completion, AC0 contains the number of bytes (or the two's complement of the number of bytes) left to compare in string 2. AC1 contains the return code as shown in the list above. AC2 contains a byte pointer either to the failing byte in string 2 (if an inequality was found) or to the byte following string 2 (if string 2 was exhausted). AC3 contains a byte pointer either to the failing byte in string 1 (if an inequality was found) or to the byte following string 1 (if string 1 was exhausted). Carry is unchanged and *overflow* is 0.

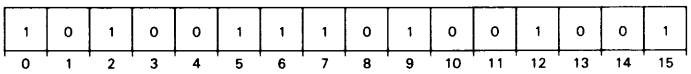
If AC0 and AC1 both contain zero (both string 1 and string 2 have length zero), the instruction returns 0 in AC1.

If the two strings are of unequal length, the instruction *fakes* space characters <040_8> in place of bytes from the exhausted string and continues the comparison.

NOTE: *The original contents of AC2 and AC3 must be valid byte pointers to an area in the user's address space. If they are invalid, a protection fault occurs, even if no bytes are to be compared. AC1 contains the code 4.*

Wide Character Move Until True

WCMT



Under control of the four accumulators, moves a string of bytes from one area of memory to another until either a table-specified delimiter character is encountered or the source string is exhausted.

The instruction copies the string one byte at a time. Before it moves a byte, the instruction uses that byte's value to determine if it is a delimiter. It treats the byte as an unsigned 8-bit binary integer (in the range $0–255_{10}$) and uses it as a bit index into a 256-bit delimiter table. If the indexed bit in the delimiter table is zero, the byte pending is not a delimiter and the instruction copies it from the source string to the destination string. If the indexed bit in the delimiter table is 1, the byte pending is a delimiter; the instruction does not copy it and the instruction terminates.

The instruction processes both strings in the same direction, either from lowest memory locations to highest (*ascending order*) or from highest memory locations to lowest (*descending order*). Processing continues until there is a delimiter or the source string is exhausted. The four accumulators contain parameters passed to the instruction.

AC0 contains the address (word address), possibly indirect, of the start of the 256-bit (16-word) delimiter table.

AC1 specifies the length of the strings and the direction of processing. If the source string is to be moved to the destination field in ascending order, AC1 contains the unsigned value of the number of bytes in the source string. If the source string is to be moved to the destination field in descending order, AC1 contains the two's complement of the number of bytes in the source string.

AC2 contains a byte pointer to the first byte to be written in the destination field. When the process is performed in ascending order, AC2 points to the lowest byte in the destination field. When the process is performed in descending order, AC2 points to the highest byte in the destination field.

AC3 contains a byte pointer to the first byte to be processed in the source string. When the process is performed in ascending order, AC3 points to the lowest byte in the source string. When the process is performed in descending order, AC3 points to the highest byte in the source string.

The fields may overlap in any way. However, the instruction moves bytes one at a time, so certain types of overlap may produce unusual side effects.

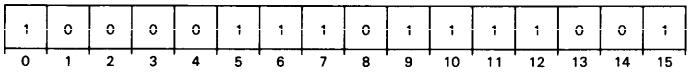
Upon completion, AC0 contains the resolved address of the translation table and AC1 contains the number of bytes (or the two's complement of the number of bytes) that were not moved. AC2 contains a byte pointer to the byte following the last byte written

in the destination field. AC3 contains a byte pointer either to the delimiter or to the first byte following the source string. The value of carry is indeterminate and *overflow* is 0.

NOTES: *The original contents of AC0, AC2, and AC3 must be valid byte pointers to an area in the user's address space. If they are invalid, a protection fault occurs, even if no bytes are to be stored. AC1 contains the code 4.*

If AC2=AC3, no bytes are written. The string is scanned for a delimiter.

Wide Character Move
WCMV



Under control of the four 32-bit accumulators, moves a string of bytes from one area of memory to another and returns a value in carry reflecting the relative lengths of source and destination strings.

The instruction copies the source string to the destination field, one byte at a time. The four accumulators contain parameters passed to the instruction. Two accumulators specify the starting address, number of bytes to be copied, and the direction of processing (ascending or descending addresses) for each field.

AC0 specifies the length and direction of processing for the destination field. If the field is to be processed from its lowest memory location to the highest, AC0 contains the unsigned value of the number of bytes in the destination field. If the field is to be processed from its highest memory location to the lowest, AC0 contains the two's complement of the number of bytes in the destination field.

AC1 specifies the length and direction of processing for the source string. If the string is to be processed from its lowest memory location to the highest, AC1 contains the unsigned value of the number of bytes in the source string. If the field is to be processed from its highest memory location to the lowest, AC1 contains the two's complement of the number of bytes in the source string.

AC2 contains a byte pointer to the first byte to be written in the destination field. When the field is written in ascending order, AC2 points to the lowest byte. When the field is written in descending order, AC2 points to the highest byte.

AC3 contains a byte pointer to the first byte copied in the source string. When the field is copied in ascending order, AC3 points to the lowest byte. When the field is copied in descending order, AC3 points to the highest byte.

The fields may overlap in any way. However, the instruction moves bytes one at a time, so certain types of overlap may produce unusual side effects.

Upon completion, AC0 contains 0 and AC1 contains the number of bytes (or the two's complement of the number of bytes) left to fetch from the source field. AC2 contains a byte pointer to the byte following the destination field; and AC3 contains a byte pointer to the byte following the last byte fetched from the source field. The value of carry is indeterminate and *overflow* is 0.

If the source field is shorter than the destination field, the instruction pads the destination field with space characters $<040_8>$. If the source field is longer than the destination field, the instruction terminates when the destination field is filled and returns the value 1 in carry; otherwise, carry is unchanged.

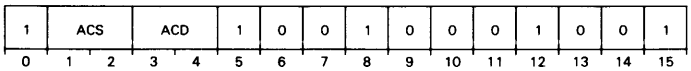
NOTES: *If AC0 contains the number 0 at the beginning of this instruction, no bytes are fetched and none are stored. If AC1 is 0 at the beginning of this instruction, the destination field is filled with space characters; note that AC3 must still contain a valid byte pointer.*

The original values of AC2 and AC3 must be valid byte pointers to an area in the user's address space. If they are invalid, a protection fault occurs, even if no bytes are to be moved. AC1 contains the code 4.

No inward ring crossing can occur for backward moves. The processor checks for this action before execution begins and executes a protection fault if it occurs.

Wide Count Bits

WCOB *acs,acd*

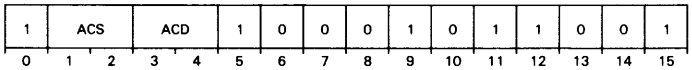


Counts and adds the number of ones in an accumulator to another accumulator.

Counts the number of bits in ACS whose value is 1. Adds the count of nonzero bits to the 32-bit, signed contents of ACD. The contents of ACS remain unchanged, unless ACS and ACD are the same accumulator. Carry is unchanged and *overflow* is 0.

Wide Complement

WCOM *acs,acd*

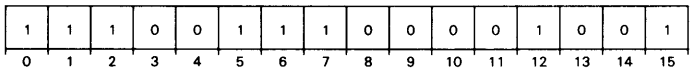


Forms the one's complement of an integer in an accumulator.

Forms the one's complement of the 32-bit fixed-point integer contained in ACS and loads the result into ACD. The contents of ACS remain unchanged, unless ACS equals ACD. Carry is unchanged and *overflow* is 0.

Wide Character Scan Until True

WCST



Under control of the four accumulators, scans a string of bytes until either a table-specified delimiter character is found or the string is exhausted.

The instruction scans the string one byte at a time. It uses each byte's value to determine if it is a delimiter. It treats the byte as an unsigned 8-bit binary integer (in the range $0–255_{10}$) and uses it as a bit index into a 256-bit delimiter table. If the indexed bit in the delimiter table is zero, the byte is not a delimiter and the instruction processes the next byte. If the indexed bit in the delimiter table is 1, the byte is a delimiter; the instruction terminates.

The instruction processes the string either from lowest memory locations to highest (*ascending order*) or from highest memory locations to lowest (*descending order*). Processing continues until there is a delimiter or the scan string is exhausted. Three accumulators contain parameters passed to the instruction.

AC0 contains the address (word address), possibly indirect, of the start of the 256-bit (16-word) delimiter table.

AC1 specifies the length of the string and the direction of processing. If the string is scanned in ascending order, AC1 contains the unsigned value of the number of bytes in the string. If the string is scanned in descending order, AC1 contains the two's complement of the number of bytes in the string.

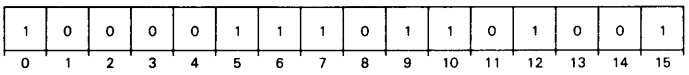
AC3 contains a byte pointer to the first byte to be processed in the string. When the process is performed in ascending order, AC3 points to the lowest byte in the string. When the process is performed in descending order, AC3 points to the highest byte in the string.

Upon completion, AC0 contains the resolved address of the translation table and AC1 contains the number of bytes (or the two's complement of the number of bytes) that were not scanned. AC3 contains a byte pointer either to the delimiter or to the first byte following the string. The value of carry is indeterminate and *overflow* is 0.

NOTE: *The original contents of AC0 and AC3 must be valid byte pointers to an area in the user's address space. If they are invalid, a protection fault occurs, even if no bytes are to be scanned. AC1 contains the code 4.*

Wide Character Translate

WCTR



Under control of the four accumulators, translates a string of bytes from one data representation to another and either moves it to another area of memory or compares it to a second translated string.

The instruction operates in two modes: translate and move, and translate and compare.

When operating in translate and move mode, the instruction translates each byte in string 1 and places it in a corresponding position in string 2. Translation is performed by using each byte as an 8-bit index into a 256-byte translation table. The byte addressed by the index then becomes the translated value.

When operating in translate and compare mode, the instruction translates each byte in string 1 and string 2 as described above and compares the translated values. Each translated byte is treated as an unsigned 8-bit binary quantity in the range $0–255_{10}$. If two translated bytes are not equal, the string whose byte has the smaller numerical value is, by definition, the *lower valued* string. Both strings remain unchanged.

AC0 specifies the address, either direct or indirect, of a word which contains a byte pointer to the first byte in the 256-byte translation table.

AC1 specifies the length of the two strings and the mode of processing. If string 1 is to be processed in translate and move mode, AC1 contains the two's complement of the number of bytes in the strings. If the strings are to be processed in translate and compare mode, AC1 contains the unsigned value of the number of bytes in the strings. Both strings are processed from lowest memory address to highest.

AC2 contains a 32-bit byte pointer to the first byte in string 2.

AC3 contains a 32-bit byte pointer to the first byte in string 1.

Upon completion of a translate and move operation, AC0 contains the address of the word which contains the byte pointer to the translation table and AC1 contains 0. AC2 contains a byte pointer to the byte following string 2 and AC3 contains a byte pointer to the byte following string 1. The value of carry is unchanged and *overflow* is 0.

Upon completion of a translate and compare operation, AC0 contains the address of the word which contains the byte pointer to the translation table. AC1 contains a return code as calculated in the list below. AC2 contains a byte pointer to either the failing byte in string 2 (if an inequality was found) or the byte following string 2 if the strings were identical. AC3 contains a byte pointer to either the failing byte in string 1 (if an inequality was found) or the byte following string 1 if the strings were identical. The value of carry is unchanged and *overflow* is 0.

Code	Result
-1	Translated value of string 1 < translated value of string 2
0	Translated value of string 1 = translated value of string 2
+1	Translated value of string 1 > translated value of string 2

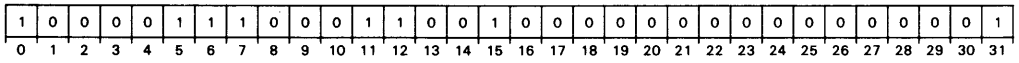
If the length of both string 1 and string 2 is zero, the compare option returns a 0 in AC1.

The fields may overlap in any way. However, processing is done one character at a time, so unusual side effects may be produced by certain types of overlap.

NOTE: *The original contents of AC0, AC2, and AC3 must be valid byte pointers to an area in the user's address space. If they are invalid, a protection fault occurs, even if no bytes are to be moved or compared. AC1 contains the code 4.*

Wide Decimal Compare

WDCMP



Compares two decimal strings of types 0 through 5 to determine which string is larger.

AC0 must contain the data type indicator describing decimal string Arg1.

AC1 must contain the data type indicator describing decimal string Arg2.

AC2 must contain a 32-bit byte pointer to the high-order byte of decimal string Arg1 in memory.

AC3 must contain a 32-bit byte pointer to the high-order byte of decimal string Arg2 in memory.

Upon completion, AC0, AC2, AC3, and the decimal strings are unchanged. Carry is unchanged and overflow is 0. AC1 holds a return code as follows:

AC1 = -1 (Arg1 is less than Arg2)

AC1 = 0 (Arg1 is equal to Arg2)

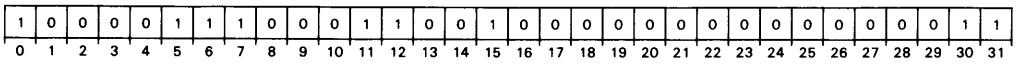
AC1 = 1 (Arg1 is greater than Arg2)

NOTES: *If the scale factors in the data type indicators of Arg1 and Arg2 are not equal, then the results are undefined.*

Any digit position which is not actually present in either argument's digit string, but needs to be considered in the comparison will be treated as being zero.

Wide Decimal Decrement

WDDEC



Subtracts one from a decimal string of type 0 through 5.

AC1 must contain the data type indicator describing the integer.

AC3 must contain a 32-bit byte pointer to the high-order byte of the decimal string in memory.

Upon completion, the accumulators are unchanged. Overflow is 0.

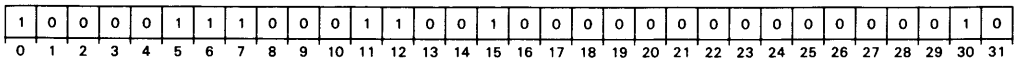
NOTES: *If the scale factor in the data type indicator is not equal to 0, then the result is undefined.*

If the decrement overflows the decimal string, Carry will be set to 1, otherwise Carry is 0. The low order of the result is stored and the high order is ignored.

If the result is negative and the data-type is 4 (unsigned), then any negative sign of the result is ignored and the absolute value of the result is stored.

Wide Decimal Increment

WDINC



Adds one to a decimal string of type 0 through 5.

AC1 must contain the data type indicator describing the integer.

AC3 must contain a 32-bit byte pointer to the high-order byte of the decimal string in memory.

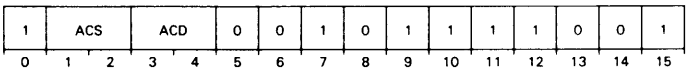
Upon completion, the accumulators are unchanged. Overflow is 0.

NOTES: *If the scale factor in the data type indicator is not equal to 0, then the result is undefined.*

If the increment overflows the decimal string, Carry will be set to 1, otherwise Carry is 0. The low order of the result is stored and the high order is ignored.

Wide Divide

WDIV *acs,acd*

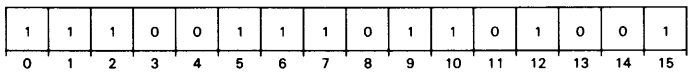


Divides an integer contained in an accumulator by an integer contained in another accumulator.

The instruction sign extends the signed, 32-bit integer contained in ACD to 64 bits. Divides this integer by the signed, 32-bit integer contained in ACS. If the quotient is within the range −2,147,483,648 to +2,147,483,647 inclusive, loads the quotient in ACD. If the result is not within this range or if ACS is zero, sets *overflow* to 1 and does not load the quotient into ACD; otherwise, *overflow* is 0. The contents of ACS and carry remain unchanged.

Wide Signed Divide

WDIVS



Divides an integer contained in AC0 and AC1 by an integer contained in AC2.

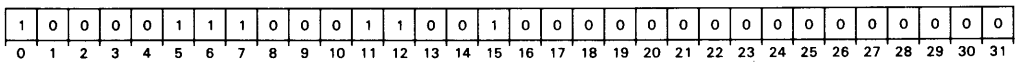
AC0 and AC1 contain a 64-bit, signed integer. AC0 contains the high-order bits.

The instruction divides the 64-bit, signed integer contained in AC0 and AC1 by the 32-bit, signed integer contained in AC2. If the quotient is within the range −2,147,483,648 to +2,147,483,647 inclusive, then places the 32-bit quotient in AC1 and the remainder in AC0. If the quotient is not within this range or if AC2 is zero, AC0 and AC1 remain unchanged and *overflow* is 1; otherwise, *overflow* is 0. AC2 and carry will always remain unchanged.

NOTE: *Zero remainders are always positive. All other remainders have the same sign as the dividend.*

Wide Decimal Move

WDMOV



Moves and scales the integer decimal string Source into the integer decimal string Destination. Source and Destination strings are of data type 0 through 5.

Under control of the four accumulators, the instruction translates and moves the decimal Source string into the decimal Destination string possibly changing commercial format while also performing scaling.

AC0 must contain the data type indicator describing Source.

AC1 must contain the data type indicator describing Destination.

AC2 must contain a 32-bit byte pointer to the high-order byte of Source in memory.

AC3 must contain a 32-bit byte pointer to the high-order byte of Destination in memory.

The accumulators and the Source string are unchanged by this instruction. If the Source string has one or more non-zero digits in positions of greater significance than the Destination string can represent, the Destination string will receive whatever digits of the Source string it is capable of representing and Carry will be set to one, otherwise Carry is unchanged. Overflow is 0.

NOTES: *Any digits of the Source string in positions of lesser significance than the Destination string can represent will be ignored.*

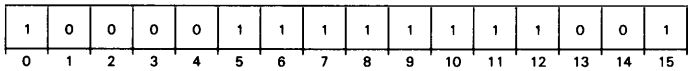
The integer placed in Destination will be zero-extended if necessary to fill the decimal string.

If Source is negative zero, the Destination will not be changed to positive zero, i.e. the sign of Destination is given the sign of Source.

If Source is negative and Destination is data-type 4 (unsigned), then the sign of Source is ignored and the absolute value of Source is moved to Destination.

If the Source and Destination strings overlap in memory in any way the results are undefined.

Pop Context Block
WDPOP



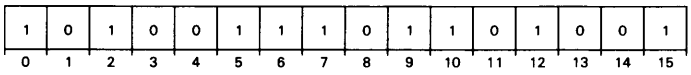
Restores the state of the machine to what it was at the time of the last page fault.

The instruction uses the information pointed to by the context block pointer in page zero of Segment 0 to restore the state of the CPU to that of the time of the last page fault. Execution of the interrupted program resumes before, during, or after the instruction that caused the fault, depending on the instruction type and how far it had proceeded before the fault. Carry is loaded from the context block and *overflow* is 0.

NOTE: *This is a privileged instruction.*

Refer to the functional characteristics manual for your machine.

Wide Edit
WEDIT



Converts a decimal source number from either packed or unpacked form to a string of bytes under the control of an edit subprogram. This subprogram can perform many different operations on the number and its destination field including leading zero suppression, leading or trailing signs, floating fill characters, punctuation control, and insertion of text into the destination field. The instruction also performs operations on alphanumeric data if you specify data type 4.

Upon entry to the *Wedit* instruction, the accumulators contain the following data:

- AC0 contains a 32-bit byte pointer to the first opcode of the *Wedit* subprogram in the current segment,
- AC1 contains a data-type indicator describing the number to be processed,
- AC2 contains a 32-bit byte pointer to the first byte of the destination field,
- AC3 contains a 32-bit byte pointer to the first byte of the source field.

The fields may overlap in any way. However, the instruction processes characters one at a time, so unusual side effects may be produced by certain types of overlap.

The instruction maintains two flags and three indicators or pointers. The flags are the Significance Trigger (*T*) and the Sign flag (*S*). The three indicators are the Source Indicator (SI), the Destination Indicator (DI), and the opcode Pointer (P).

At the start of execution, the *Wedit* instruction sets *T* to 0. When the instruction manipulates the first nonzero digit, it sets *T* to 1 (unless an edit opcode specifies otherwise).

The instruction sets *S* to reflect the sign of the number currently being processed. If the number is positive, the instruction sets *S* to 0. If the number is negative, the instruction sets *S* to 1.

Each of the three indicators is 32 bits wide and contains a byte pointer to the *current* byte in each respective area. At the start of execution, the *Wedit* instruction sets SI to the value contained in AC3 (the starting address of the source string). It also sets DI to the value contained in AC2 (the starting address of the destination string) and P to the value contained in AC0 (a pointer to the first *Wedit* opcode).

During execution, the subprogram can test and modify *S* and *T*, as well as modify SI, DI and P.

When execution begins with a signed source number, the instruction checks the sign of the source number for validity. If the sign is invalid, the instruction ends. If the sign is valid, execution continues with the *Wedit* subprogram.

The subprogram is made up of 8-bit opcodes followed by one or more 8-bit operands. The byte pointer contained in P acts as the *program counter* for the subprogram. The subprogram proceeds sequentially until a branching operation occurs — much the same way programs are processed. Unless instructed to do otherwise, the *Wedit* instruction updates P after each operation to point to the next sequential opcode. The instruction continues to process 8-bit opcodes until directed to stop by the **DEND** opcode.

NOTE: *The WEDIT instruction considers the subprogram as data, and does not check for execute protection on it.*

Upon successful termination, carry contains *T*; AC0 contains P, which points to the next opcode to be processed; AC1 is undefined; AC2 contains DI, which points to the next destination byte; and AC3 contains SI, which points to the next source byte. *Overflow* is 0.

NOTES: *If SI references bytes not contained in the source number, then the instruction supplies zeroes for future manipulations. The instruction will use these zeroes for all subsequent operations, even if SI later references bytes contained by the source number.*

Opcodes that move numeric data may perform special actions. Opcodes that move non-numeric data copy characters exactly into the destination string.

The Wedit *instruction places information on the wide stack. Therefore, the stack must be set up and have at least 16 words available for use.*

If an interrupt occurs during the Wedit *instruction, the instruction places restart information on the stack and in the accumulators, and sets bit 2 of the PSR to 1.*

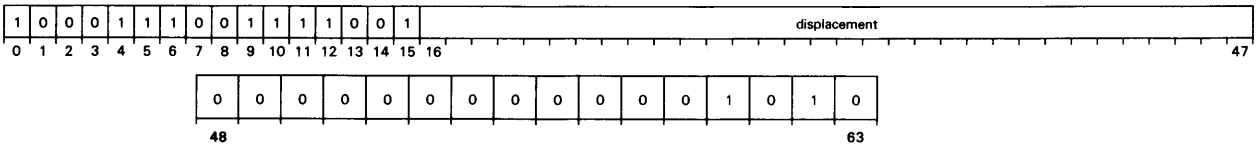
If bit 2 of the PSR contains a 1, then the Wedit *instruction assumes it is restarting from an interrupt. Make sure you do not set this bit under any other circumstances.*

Many of the *Wedit* opcodes use the symbol *j*. This symbol represents a number; when *j* is greater than or equal to zero, it specifies the number of characters the instruction should process. When *j* is less than zero, it represents a pointer into the wide stack. The pointer references a stack word that denotes the number of characters the instruction should process. The number on the stack is at address WSP + 2 + 2*j.

A *Wedit* operation that processes numeric data (e.g., **DMVN**) skips a leading or trailing sign code it encounters; similarly, such an operation converts a high-order or low-order sign to its correct numeric equivalent.

Floating-Point Arccosine Double

WFACOSD displacement



Computes the arccosine of the 64-bit floating-point value in FPAC0, and places the result (in radians) in FPAC0.

> **NOTE:** *If hardware support is presently unavailable, the processor performs an LPSHJ instruction function and computes the effective address, E (using the displacement as a non-indirectable PC-relative offset). Uses E as the address of a routine in a run-time library which performs the arccosine function. Return from this software emulator is by the WPOPJ instruction. The effective address must be in the current ring.*

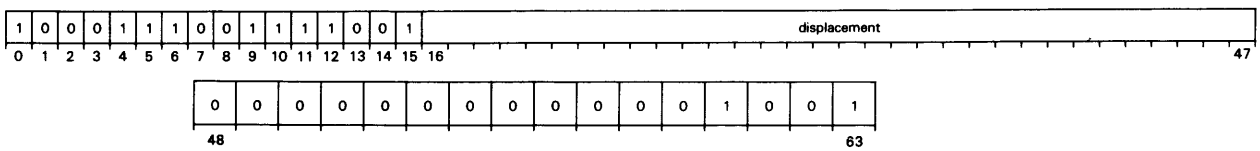
Returns the floating-point result to FPAC0. Updates the Z and N flags of the floating-point status register to reflect the new contents of FPAC0.

Upon completion, the contents of FPAC1, FPAC2, and FPAC3 are undefined.

> **NOTE:** *If the absolute value of the input number in FPAC0 is greater than one, then the processor sets the INV bit in the FPSR to one and returns error code three to the INP bits of the FPSR.*
>
> *This instruction is one of the optional floating-point intrinsic instructions (IIS).*

Floating-Point Arccosine Single

WFACOSS displacement



Computes the arccosine of the 32-bit floating-point value in FPAC0, and places the result (in radians) in FPAC0.

> **NOTE:** *If hardware support is presently unavailable, the processor performs an LPSHJ instruction function and computes the effective address, E (using the displacement as a non-indirectable PC-relative offset). Uses E as the address of a routine in a run-time library which performs the arccosine function. Return from this software emulator is by the WPOPJ instruction. The effective address must be in the current ring.*

Returns the floating-point result (in radians) to FPAC0 bits 0-31 and sets bits 32-63 to 0. Updates the Z and N flags of the floating-point status register to reflect the new contents of FPAC0.

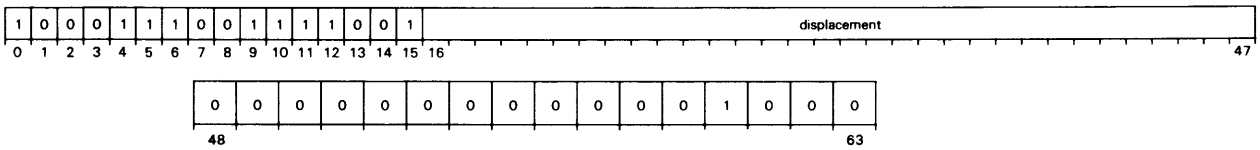
Upon completion, the contents of FPAC1, FPAC2, and FPAC3 are undefined.

NOTE: *If the absolute value of the input number in FPAC0 is greater than one, then the processor sets the INV bit in the FPSR to one and returns error code three to the INV bits of the FPSR.*

This instruction is one of the optional floating-point intrinsic instructions (IIS).

Floating-Point Arcsine Double

WFASIND displacement



Computes the arcsine of the 64-bit floating-point value in FPAC0, and places the result (in radians) in FPAC0.

NOTE: *If hardware support is presently unavailable, the processor performs an LPSHJ instruction function and computes the effective address, E (using the displacement as a non-indirectable PC-relative offset). Uses E as the address of a routine in a run-time library which performs the arcsine function. Return from this software emulator is by the WPOPJ instruction. The effective address must be in the current ring.*

Returns the floating-point result to FPAC0. Updates the Z and N flags of the floating-point status register to reflect the new contents of FPAC0.

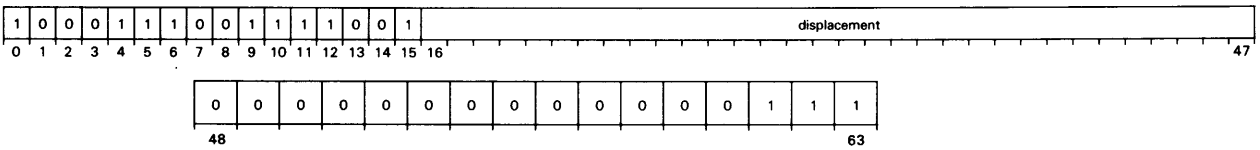
Upon completion, the contents of FPAC1, FPAC2, and FPAC3 are undefined.

NOTE: *If the absolute value of the input number in FPAC0 is greater than one, then the processor sets the INV bit in the FPSR to one and returns error code three to the INP bits of the FPSR.*

This instruction is one of the optional floating-point intrinsic instructions (IIS).

Floating-Point Arcsine Single

WFASINS displacement



Computes the arcsine of the 32-bit floating-point value in FPAC0, and places the result (in radians) in FPAC0.

NOTE: *If hardware support is presently unavailable, the processor performs an LPSHJ instruction function and computes the effective address, E (using the displacement as a non-indirectable PC-relative offset). Uses E as the address of a routine in a run-time library which performs the arcsine function. Return from this software emulator is by the WPOPJ instruction. The effective address must be in the current ring.*

Returns the floating-point result (in radians) to FPAC0 bits 0-31 and sets bits 32-63 to 0. Updates the Z and N flags of the floating-point status register to reflect the new contents of FPAC0.

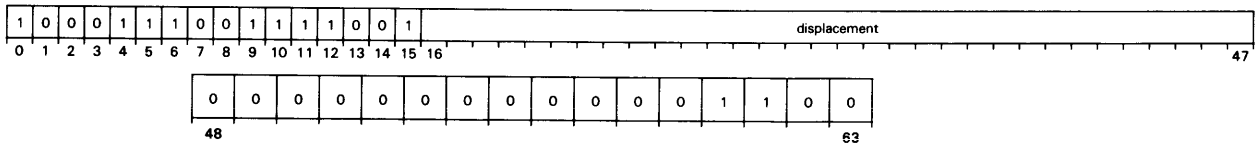
Upon completion, the contents of FPAC1, FPAC2, and FPAC3 are undefined.

NOTE: *If the absolute value of the input number in FPAC0 is greater than one, then the processor sets the INV bit in the FPSR to one and returns error code three to the INP bits of the FPSR.*

This instruction is one of the optional floating-point intrinsic instructions (IIS).

Floating-Point Arctangent Double

WFATAND displacement



Computes the arctangent of the 64-bit floating-point value in FPAC0, and places the result (in radians) in FPAC0.

NOTE: *If hardware support is presently unavailable, the processor performs an LPSHJ instruction function and computes the effective address, E (using the displacement as a non-indirectable PC-relative offset). Uses E as the address of a routine in a run-time library which performs the arctangent function. Return from this software emulator is by the WPOPJ instruction. The effective address must be in the current ring.*

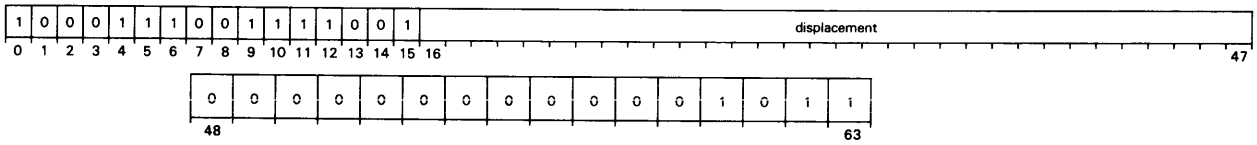
Returns the floating-point result to FPAC0. Updates the Z and N flags of the floating-point status register to reflect the new contents of FPAC0.

Upon completion, the contents of FPAC1, FPAC2, and FPAC3 are undefined.

NOTE: *This instruction is one of the optional floating-point intrinsic instructions (IIS).*

Floating-Point Arctangent Single

WFATANS displacement



Computes the arctangent of the 32-bit floating-point value in FPAC0, and places the result (in radians) in FPAC0.

NOTE: *If hardware support is presently unavailable, the processor performs an LPSHJ instruction function and computes the effective address, E (using the displacement as a non-indirectable PC-relative offset). Uses E as the address of a routine in a run-time library which performs the arctangent function. Return from this software emulator is by the WPOPJ instruction. The effective address must be in the current ring.*

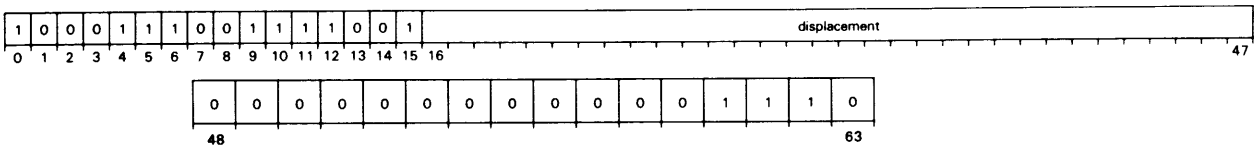
Returns the floating-point result (in radians) to FPAC0 bits 0-31 and sets bits 32-63 to 0. Updates the Z and N flags of the floating-point status register to reflect the new contents of FPAC0.

Upon completion, the contents of FPAC1, FPAC2, and FPAC3 are undefined.

NOTE: *This instruction is one of the optional floating-point intrinsic instructions (IIS).*

Floating-Point Arctangent Double (two-accumulator)

WFATN2D displacement



Computes the arctangent of the quotient of two 64-bit floating-point values (FPAC0 divided by FPAC1), and places the result with correct quadrature (in radians) in FPAC0.

NOTE: *If hardware support is presently unavailable, the processor performs an LPSHJ instruction function and computes the effective address, E (using the displacement as a non-indirectable PC-relative offset). Uses E as the address of a routine in a run-time library which performs the arctangent function. Return from this software emulator is by the WPOPJ instruction. The effective address must be in the current ring.*

FPAC0 contains the Y value; FPAC1 contains the X value.

Returns the floating-point result to FPAC0. Updates the Z and N flags of the floating-point status register to reflect the new contents of FPAC0.

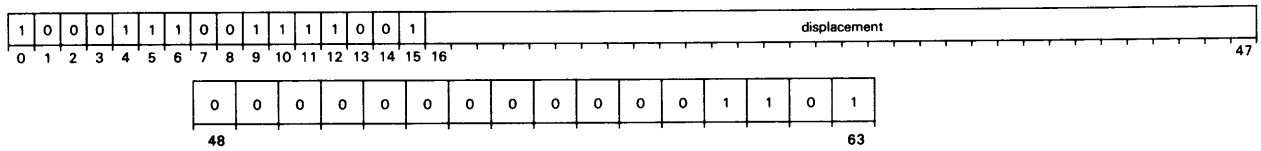
Upon completion, the contents of FPAC1, FPAC2, and FPAC3 are undefined.

NOTE: *If the value in FPAC1 is zero, then the processor sets the INV bit in the FPSR to one and returns error code seven to the INP bits of the FPSR*

This instruction is one of the optional floating-point intrinsic instructions (IIS).

Floating-Point Arctangent Single (two-accumulator)

WFATN2S displacement



Computes the arctangent of the quotient of two 32-bit floating-point values (FPAC0 divided by FPAC1) and places the result with correct quadrature (in radians) in FPAC0.

NOTE: *If hardware support is presently unavailable, the processor performs an LPSHJ instruction function and computes the effective address, E (using the displacement as a non-indirectable PC-relative offset). Uses E as the address of a routine in a run-time library which performs the arctangent function. Return from this software emulator is by the WPOPJ instruction. The effective address must be in the current ring.*

FPAC0 contains the Y value; FPAC1 contains the X value.

Returns the floating-point result (in radians) to FPAC0 bits 0-31 and sets bits 32-63 to 0. Updates the Z and N flags of the floating-point status register to reflect the new contents of FPAC0.

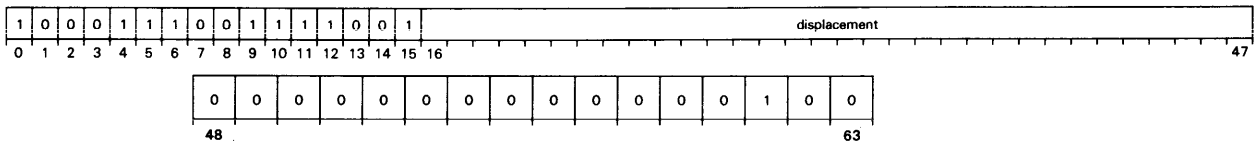
Upon completion, the contents of FPAC1, FPAC2, and FPAC3 are undefined.

NOTE: *If the value in FPAC1 is zero, then the processor sets the INV bit in the FPSR to one and returns error code seven to the INP bits of the FPSR.*

This instruction is one of the optional floating-point intrinsic instructions (IIS).

Floating-Point Cosine Double

WFCOSD displacement



Computes the cosine of the 64-bit floating-point value (in radians) in FPAC0, and places the result in FPAC0.

> **NOTE:** *If hardware support is presently unavailable, the processor performs an LPSHJ instruction function and computes the effective address, E (using the displacement as a non-indirectable PC-relative offset). Uses E as the address of a routine in a run-time library which performs the cosine function. Return from this software emulator is by the WPOPJ instruction. The effective address must be in the current ring.*

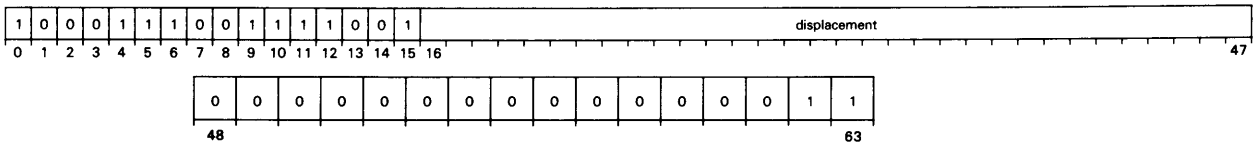
Returns the floating-point result to FPAC0. Updates the Z and N flags of the floating-point status register to reflect the new contents of FPAC0.

Upon completion, the contents of FPAC1, FPAC2, and FPAC3 are undefined.

> **NOTE:** *This instruction is one of the optional floating-point intrinsic instructions (IIS).*

Floating-Point Cosine Single

WFCOSS displacement



Computes the cosine of the 32-bit floating-point value (in radians) in FPAC0, and places the result in FPAC0.

> **NOTE:** *If hardware support is presently unavailable, the processor performs an LPSHJ instruction function and computes the effective address, E (using the displacement as a non-indirectable PC-relative offset). Uses E as the address of a routine in a run-time library which performs the cosine function. Return from this software emulator is by the WPOPJ instruction. The effective address must be in the current ring.*

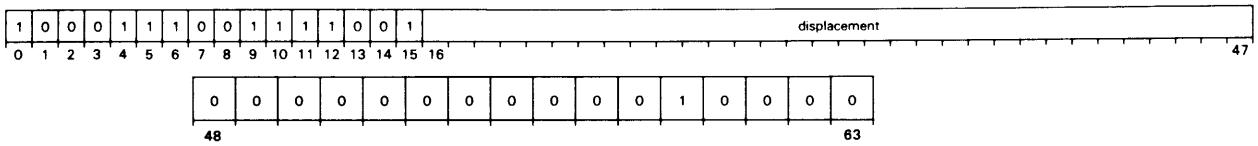
Returns the floating-point result to FPAC0 bits 0-31 and sets bits 32-63 to 0. Updates the Z and N flags of the floating-point status register to reflect the new contents of FPAC0.

Upon completion, the contents of FPAC1, FPAC2, and FPAC3 are undefined.

> **NOTE:** *This instruction is one of the optional floating-point intrinsic instructions (IIS).*

Floating-Point Exponential Double

WFEXPD displacement



Raises the value e to the 64-bit floating-point power contained in FPAC0, and places the result in FPAC0.

NOTE: *If hardware support is presently unavailable, the processor performs an LPSHJ instruction function and computes the effective address, E (using the displacement as a non-indirectable PC-relative offset). Uses E as the address of a routine in a run-time library which performs the exponential function. Return from this software emulator is by the WPOPJ instruction. The effective address must be in the current ring.*

Returns the floating-point result to FPAC0. Updates the Z and N flags of the floating-point status register to reflect the new contents of FPAC0.

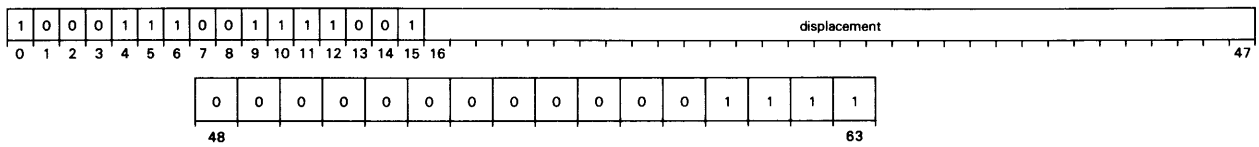
Upon completion, the contents of FPAC1, FPAC2, and FPAC3 are undefined.

NOTE: *If the input value in FPAC0 will produce a result which overflows (i.e., approximately the natural log of 16 raised to the 63rd power), then the processor sets the INV bit in the FPSR to one and returns error code five to the INP bits of the FPSR.*

This instruction is one of the optional floating-point intrinsic instructions (IIS).

Floating-Point Exponential Single

WFEXPS displacement



Raises the value e to the 32-bit floating-point power contained in FPAC0, and places the result in FPAC0.

NOTE: *If hardware support is presently unavailable, the processor performs an LPSHJ instruction function and computes the effective address, E (using the displacement as a non-indirectable PC-relative offset). Uses E as the address of a routine in a run-time library which performs the exponential function. Return from this software emulator is by the WPOPJ instruction. The effective address must be in the current ring.*

Returns the floating-point result to FPAC0 bits 0-31 and sets bits 32-63 to 0. Updates the Z and N flags of the floating-point status register to reflect the new contents of FPAC0.

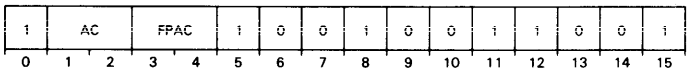
Upon completion, the contents of FPAC1, FPAC2, and FPAC3 are undefined.

NOTE: *If the input value in FPAC0 will produce a result which overflows (i.e., approximately the natural log of 16 raised to the 63rd power), then the processor sets the INV bit in the FPSR to one and returns error code five to the INP bits of the FPSR.*

This instruction is one of the optional floating-point intrinsic instructions (IIS).

Wide Fix from Floating-Point Accumulator

WFFAD *ac,fpac*



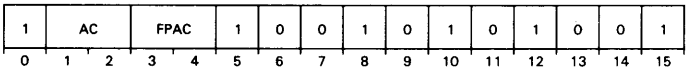
Converts the integer portion of the floating-point number contained in the specified FPAC to a 32-bit, signed, two's complement integer. Places the result in an accumulator.

If the integer portion of the number contained in FPAC is less than −2,147,483,648 or greater than +2,147,483,647, the instruction sets *MOF* in the FPSR to 1. Takes the absolute value of the integer portion of the number contained in the FPAC. Takes the 31 least significant bits of the absolute value and appends a 0 onto the leftmost bit to give a 32-bit number. If the sign of the number is negative, forms the two's complement of the 32-bit result. Places the 32-bit integer in the specified accumulator.

The FPAC and the *Z* and *N* flags of the FPSR remain unchanged.

Wide Float from Fixed-Point Accumulator

WFLAD *ac,fpac*



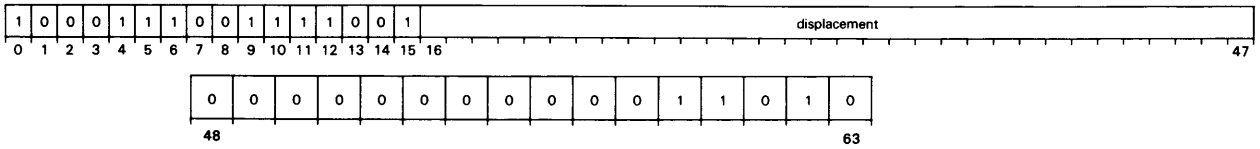
Converts the contents of a 32-bit accumulator to floating-point format and places the result in a specified FPAC.

Converts the 32-bit, signed, two's complement number contained in the specified accumulator to a double-precision floating-point number. Places the result in the specified FPAC. Updates the *Z* and *N* flags in the floating-point status register to reflect the new contents of the FPAC.

The range of numbers that can be converted is −2,147,483,648 to +2,147,483,647 inclusive.

Floating-Point Binary Logarithm Double

WFLG2D displacement



Computes the binary logarithm (log base 2) of the 64-bit floating-point number contained in FPAC0, and places the result in FPAC0.

> **NOTE:** *If hardware support is presently unavailable, the processor performs an LPSHJ instruction function and computes the effective address, E (using the displacement as a non-indirectable PC-relative offset). Uses E as the address of a routine in a run-time library which performs the binary logarithm function. Return from this software emulator is by the WPOPJ instruction. The effective address must be in the current ring.*

Returns the floating-point result to FPAC0. Updates the Z and N flags of the floating-point status register to reflect the new contents of FPAC0.

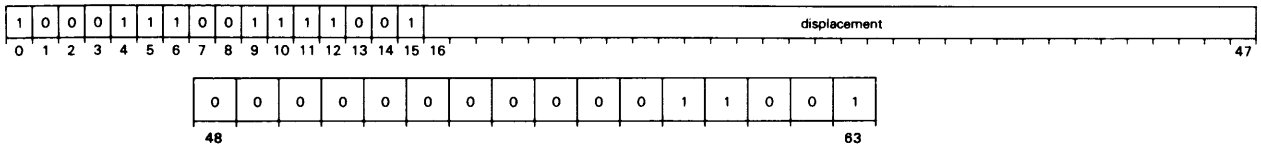
Upon completion, the contents of FPAC1, FPAC2, and FPAC3 are undefined.

NOTE: *If the input value in FPAC0 is less than or equal to 0, then the processor sets the INV bit in the FPSR to one and returns error code one to the INP bits of the FPSR.*

This instruction is one of the optional floating-point intrinsic instructions (IIS).

Floating-Point Binary Logarithm Single

WFLG2S displacement



Computes the binary logarithm (log base 2) of the 32-bit floating-point number contained in FPAC0, and places the result in FPAC0.

NOTE: *If hardware support is presently unavailable, the processor performs an LPSHJ instruction function and computes the effective address, E (using the displacement as a non-indirectable PC-relative offset). Uses E as the address of a routine in a run-time library which performs the binary logarithm function. Return from this software emulator is by the WPOPJ instruction. The effective address must be in the current ring.*

Returns the floating-point result to FPAC0 bits 0-31 and sets bits 32-63 to 0. Updates the Z and N flags of the floating-point status register to reflect the new contents of FPAC0.

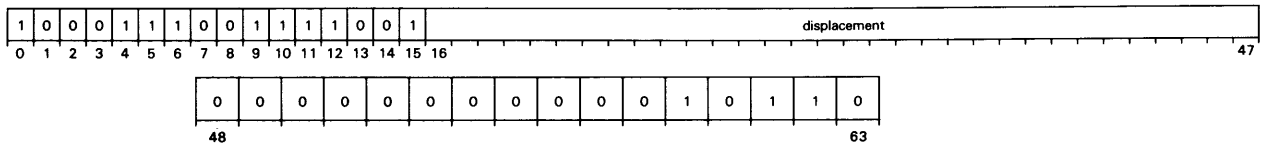
Upon completion, the contents of FPAC1, FPAC2, and FPAC3 are undefined.

NOTE: *If the input value in FPAC0 is less than or equal to 0, then the processor sets the INV bit in the FPSR to one and returns error code one to the INP bits of the FPSR.*

This instruction is one of the optional floating-point intrinsic instructions (IIS).

Floating-Point Natural Logarithm Double

WFLNGD displacement



Computes the natural logarithm (log base e) of the 64-bit floating-point number contained in FPAC0, and places the result in FPAC0.

NOTE: *If hardware support is presently unavailable, the processor performs an LPSHJ instruction function and computes the effective address, E (using the displacement as a non-indirectable PC-relative offset). Uses E as the address of a routine in a run-time library which performs the natural logarithm function. Return from this software emulator is by the WPOPJ instruction. The effective address must be in the current ring.*

Returns the floating-point result to FPAC0. Updates the Z and N flags of the floating-point status register to reflect the new contents of FPAC0.

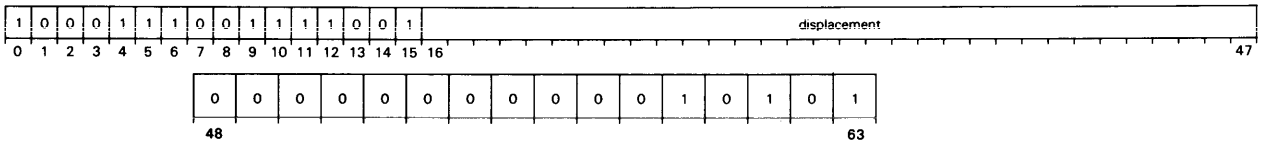
Upon completion, the contents of FPAC1, FPAC2, and FPAC3 are undefined.

NOTE: *If the input value in FPAC0 is less than or equal to 0, then the processor sets the INV bit in the FPSR to one and returns error code one to the INP bits of the FPSR.*

This instruction is one of the optional floating-point intrinsic instructions (IIS).

Floating-Point Natural Logarithm Single

WFLNGS displacement



Computes the natural logarithm (log base e) of the 32-bit floating-point number contained in FPAC0, and places the result in FPAC0.

NOTE: *If hardware support is presently unavailable, the processor performs an LPSHJ instruction function and computes the effective address, E (using the displacement as a non-indirectable PC-relative offset). Uses E as the address of a routine in a run-time library which performs the natural logarithm function. Return from this software emulator is by the WPOPJ instruction. The effective address must be in the current ring.*

Returns the floating-point result to FPAC0 bits 0-31 and sets bits 32-63 to 0. Updates the Z and N flags of the floating-point status register to reflect the new contents of FPAC0.

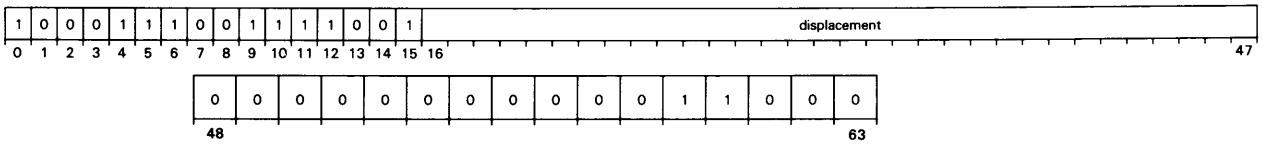
Upon completion, the contents of FPAC1, FPAC2, and FPAC3 are undefined.

NOTE: *If the input value in FPAC0 is less than or equal to 0, then the processor sets the INV bit in the FPSR to one and returns error code one to the INP bits of the FPSR.*

This instruction is one of the optional floating-point intrinsic instructions (IIS).

Floating-Point Common Logarithm Double

WFLOGD displacement



Computes the common logarithm (log base 10) of the 64-bit floating-point number contained in FPAC0, and places the result in FPAC0.

NOTE: *If hardware support is presently unavailable, the processor performs an LPSHJ instruction function and computes the effective address, E (using the displacement as a non-indirectable PC-relative offset). Uses E as the address of a routine in a run-time library which performs the common logarithm function. Return from this software emulator is by the WPOPJ instruction. The effective address must be in the current ring.*

Returns the floating-point result to FPAC0. Updates the Z and N flags of the floating-point status register to reflect the new contents of FPAC0.

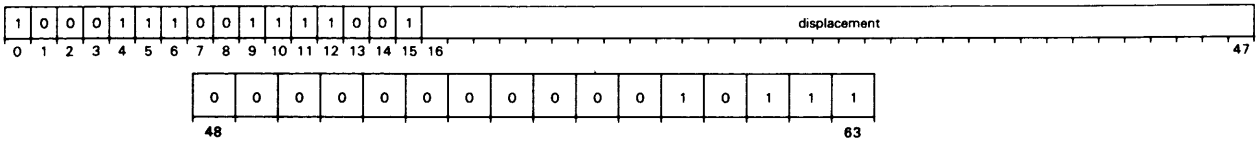
Upon completion, the contents of FPAC1, FPAC2, and FPAC3 are undefined.

NOTE: *If the input value in FPAC0 is less than or equal to 0, then the processor sets the INV bit in the FPSR to one and returns error code one to the INP bits of the FPSR.*

This instruction is one of the optional floating-point intrinsic instructions (IIS).

Floating-Point Common Logarithm Single

WFLOGS displacement



Computes the common logarithm (log base 10) of the 32-bit floating-point number contained in FPAC0, and places the result in FPAC0.

> **NOTE:** *If hardware support is presently unavailable, the processor performs an LPSHJ instruction function and computes the effective address, E (using the displacement as a non-indirectable PC-relative offset). Uses E as the address of a routine in a run-time library which performs the common logarithm function. Return from this software emulator is by the WPOPJ instruction. The effective address must be in the current ring.*

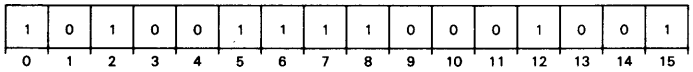
Returns the floating-point result to FPAC0 bits 0-31 and sets bits 32-63 to 0. Updates the Z and N flags of the floating-point status register to reflect the new contents of FPAC0.

Upon completion, the contents of FPAC1, FPAC2, and FPAC3 are undefined.

> **NOTE:** *If the input value in FPAC0 is less than or equal to 0, then the processor sets the INV bit in the FPSR to one and returns error code one to the INP bits of the FPSR.*
>
> *This instruction is one of the optional floating-point intrinsic instructions (IIS).*

Wide Floating-Point Pop

WFPOP



Pops the state of the floating-point unit off the wide stack.

Pops a 20-word block off the wide stack and loads the contents into the FPSR and the four FPACs. The format of the 20-word block is shown in Figure 10.9.

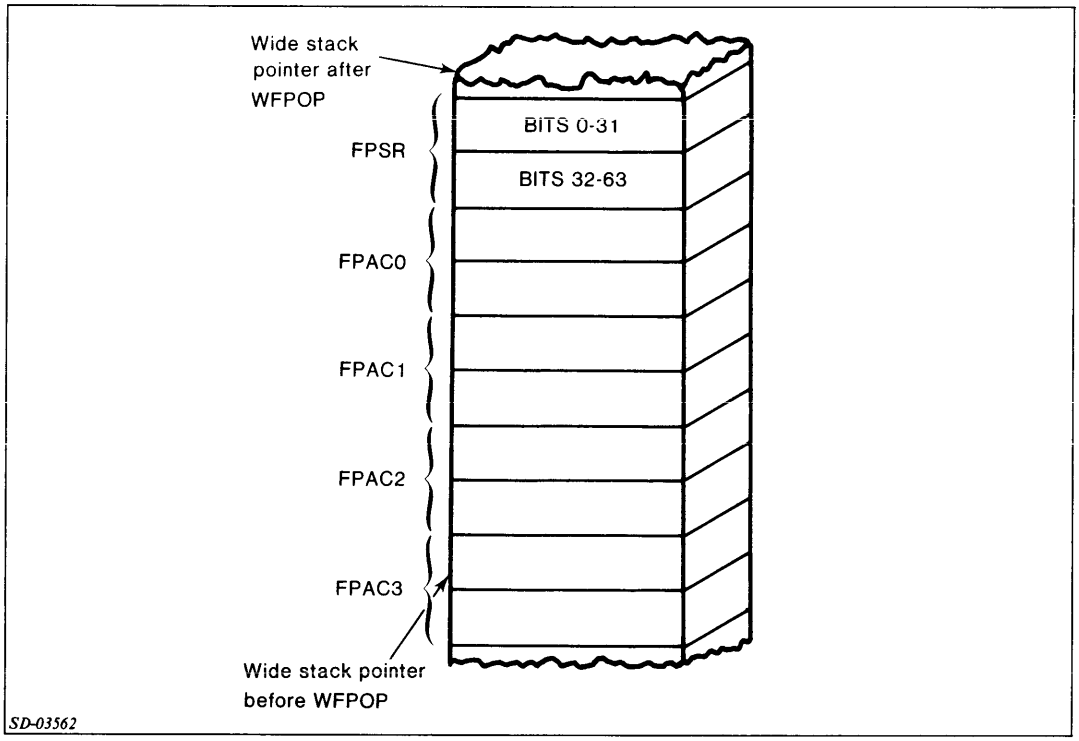


Figure 10.9 WFPOP 10 double-word

This instruction loads the FPSR as follows:

- Places bits 0–15 of the operand in bits 0–15 of the FPSR. Sets bits 16–32 of the FPSR to 0.
- If *ANY* is 0, bits 33–63 of the FPSR are undefined.
- If *ANY* is 1, the instruction places bits 28–31 of the operand in bits 28–31 (INP) of the FPSR and places bits 33–63 of the operand in bits 33–63 (FPPC) of the FPSR.

NOTE: *This instruction moves unnormalized data without change.*

This instruction does not set the ANY *flag from memory. If any of bits 1–4 are loaded as 1,* ANY *is set to 1; otherwise,* ANY *is 0.*

Bits 12–15 of the FPSR are not set from memory. These bits are the floating-point identification code and cannot be changed. Refer to the specific functional characteristics manual for the code to use.

This instruction does not initiate a floating-point trap under any conditions of the FPSR.

Wide Floating-Point Push

WFPSH

1	0	0	0	0	1	1	1	1	0	1	1	1	0	0	1
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Pushes the state of the floating-point unit onto the wide stack.

Pushes a 20-word block onto the wide stack. The block contains the contents of the FPSR and the contents of the four FPACs, as shown in Figure 10.10.

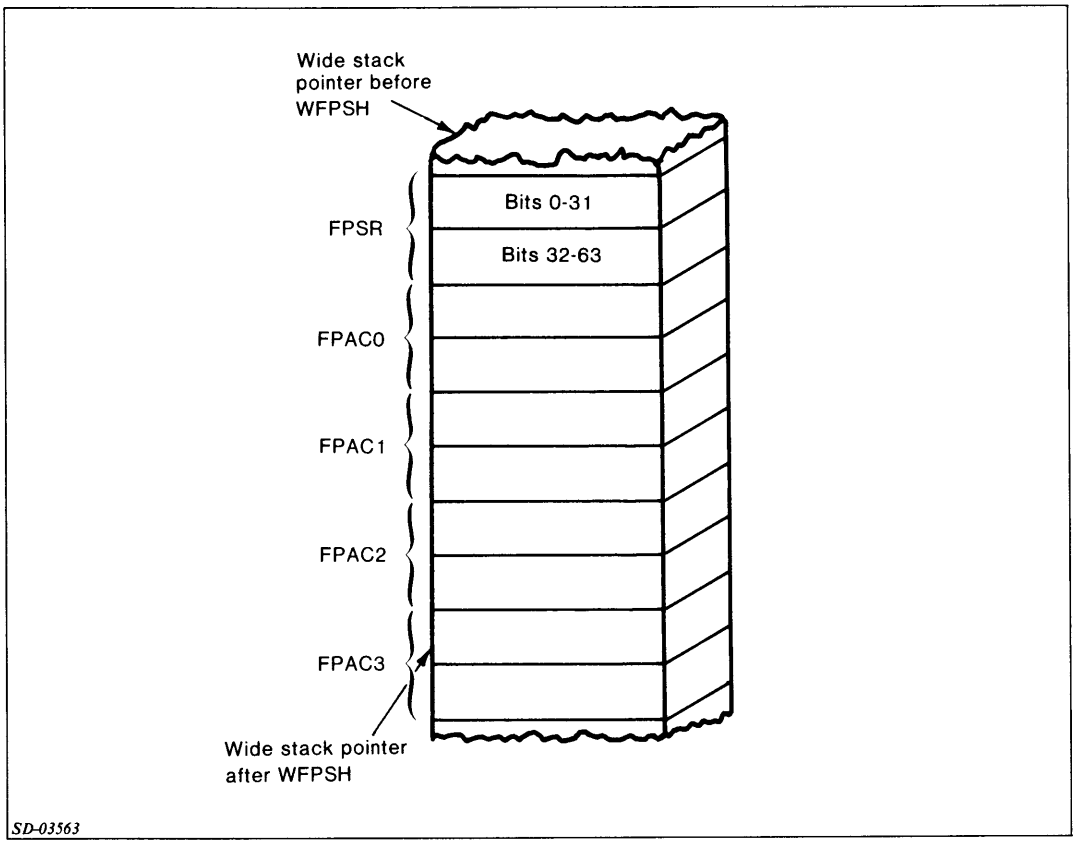


Figure 10.10 WFPSH 10 double-word block pushed

The instruction pushes the FPSR onto the stack as follows:

- Stores bits 0–15 of the FPSR in the first memory word.
- Sets bits 16–31 of the first memory double word and bit 0 of the second memory double word to 0.
- If *ANY* is 0, the contents of bits 1–31 of the second memory double word are undefined.
- If *ANY* is 1, the instruction stores bits 28–31 (INP) of the FPSR into bits 28–31 of the first memory double word and stores bits 33–63 (FPPC) of the FPSR into bits 1–31 of the second memory double word.

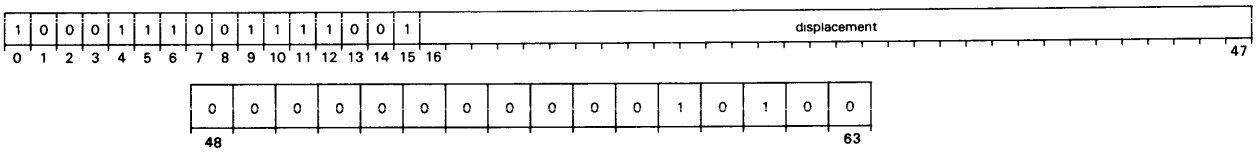
The rest of the block is pushed onto the stack after the FPSR has been pushed.

NOTES: *This instruction moves unnormalized data without change.*

This instruction does not initiate a floating-point trap under any conditions of the FPSR.

Floating-Point Power Double

WFPWRD displacement



Raises the 64-bit floating-point number contained in FPAC0 to the 64-bit floating-point power contained in FPAC1, and places the result in FPAC0.

NOTE: *If hardware support is presently unavailable, the processor performs an LPSHJ instruction function and computes the effective address, E (using the displacement as a non-indirectable PC-relative offset). Uses E as the address of a routine in a run-time library which performs the power function. Return from this software emulator is by the WPOPJ instruction. The effective address must be in the current ring.*

Returns the floating-point result to FPAC0. Updates the Z and N flags of the floating-point status register to reflect the new contents of FPAC0.

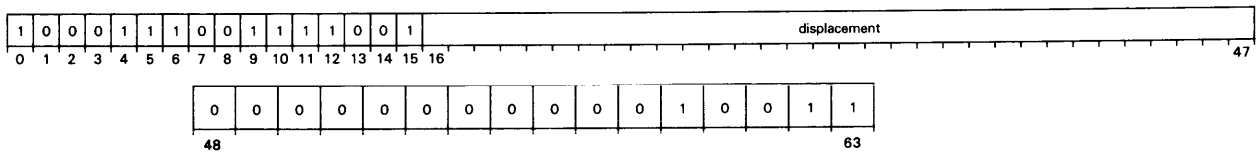
Upon completion, the contents of FPAC1, FPAC2, and FPAC3 are undefined.

NOTE: *If the input value in FPAC0 is less than 0 AND the value in FPAC1 is not equal to 0, OR the input value in FPAC0 equals 0 AND the value in FPAC1 is less than or equal to 0, then the processor sets the INV bit in the FPSR to one and returns error code four to the INP bits of the FPSR.*

This instruction is one of the optional floating-point intrinsic instructions (IIS).

Floating-Point Power Single

WFPWRS displacement



Raises the 32-bit floating-point number contained in FPAC0 to the 32-bit floating-point power contained in FPAC1, and places the result in FPAC0.

NOTE: *If hardware support is presently unavailable, the processor performs an LPSHJ instruction function and computes the effective address, E (using the displacement as a non-indirectable PC-relative offset). Uses E as the address of a routine in a run-time library which performs the power function. Return from this software emulator is by the WPOPJ instruction. The effective address must be in the current ring.*

Returns the floating-point result to FPAC0 bits 0-31 and sets bits 32-63 to 0. Updates the Z and N flags of the floating-point status register to reflect the new contents of FPAC0.

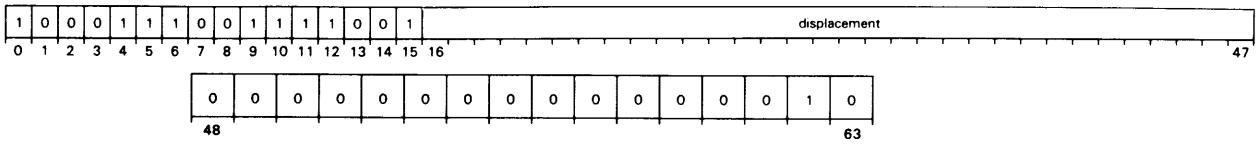
Upon completion, the contents of FPAC1, FPAC2, and FPAC3 are undefined.

NOTE: *If the input value in FPAC0 is less than 0 AND the value in FPAC1 is not equal to 0, OR the input value in FPAC0 equals 0 AND the value in FPAC1 is less than or equal to 0, then the processor sets the INV bit in the FPSR to one and returns error code four to the INP bits of the FPSR.*

This instruction is one of the optional floating-point intrinsic instructions (IIS).

Floating-Point Sine Double

WFSIND displacement



Computes the sine of the 64-bit floating-point value (in radians) in FPAC0, and places the result in FPAC0.

NOTE: *If hardware support is presently unavailable, the processor performs an LPSHJ instruction function and computes the effective address, E (using the displacement as a non-indirectable PC-relative offset). Uses E as the address of a routine in a run-time library which performs the sine function. Return from this software emulator is by the WPOPJ instruction. The effective address must be in the current ring.*

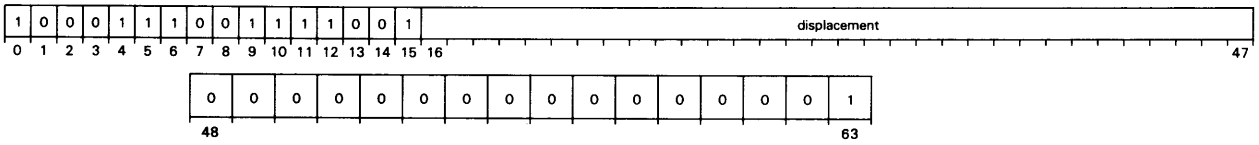
Returns the floating-point result to FPAC0. Updates the Z and N flags of the floating-point status register to reflect the new contents of FPAC0.

Upon completion, the contents of FPAC1, FPAC2, and FPAC3 are undefined.

NOTE: *This instruction is one of the optional floating-point intrinsic instructions (IIS).*

Floating-Point Sine Single

WFSINS displacement



Computes the sine of the 32-bit floating-point value (in radians) in FPAC0, and places the result in FPAC0.

NOTE: *If hardware support is presently unavailable, the processor performs an LPSHJ instruction function and computes the effective address, E (using the displacement as a non-indirectable PC-relative offset). Uses E as the address of a routine in a run-time library which performs the sine function. Return from this software emulator is by the WPOPJ instruction. The effective address must be in the current ring.*

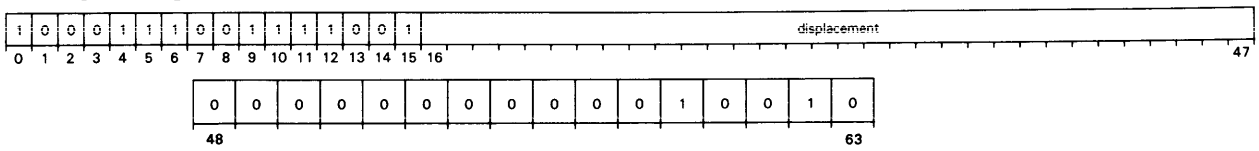
Returns the floating-point result to FPAC0 bits 0-31 and sets bits 32-63 to 0. Updates the Z and N flags of the floating-point status register to reflect the new contents of FPAC0.

Upon completion, the contents of FPAC1, FPAC2, and FPAC3 are undefined.

NOTE: *This instruction is one of the optional floating-point intrinsic instructions (IIS).*

Floating-Point Square Root Double

WFSQRD displacement



Computes the square root of the 64-bit floating-point number contained in FPAC0, and places the result in FPAC0.

> **NOTE:** *If hardware support is presently unavailable, the processor performs an LPSHJ instruction function and computes the effective address, E (using the displacement as a non-indirectable PC-relative offset). Uses E as the address of a routine in a run-time library which performs the square root function. Return from this software emulator is by the WPOPJ instruction. The effective address must be in the current ring.*

Returns the floating-point result to FPAC0. Updates the Z and N flags of the floating-point status register to reflect the new contents of FPAC0.

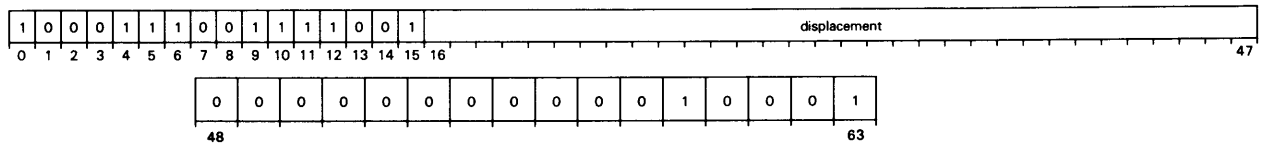
Upon completion, the contents of FPAC1, FPAC2, and FPAC3 are undefined.

> **NOTE:** *If the input value in FPAC0 is less than 0, then the processor sets the INV bit in the FPSR to one and returns error code two to the INP bits of the FPSR.*

This instruction is one of the optional floating-point intrinsic instructions (IIS).

Floating-Point Square Root Single

WFSQRS displacement



Computes the square root of the 32-bit floating-point number contained in FPAC0, and places the result in FPAC0.

> **NOTE:** *If hardware support is presently unavailable, the processor performs an LPSHJ instruction function and computes the effective address, E (using the displacement as a non-indirectable PC-relative offset). Uses E as the address of a routine in a run-time library which performs the square root function. Return from this software emulator is by the WPOPJ instruction. The effective address must be in the current ring.*

Returns the floating-point result to FPAC0 bits 0-31 and sets bits 32-63 to 0. Updates the Z and N flags of the floating-point status register to reflect the new contents of FPAC0.

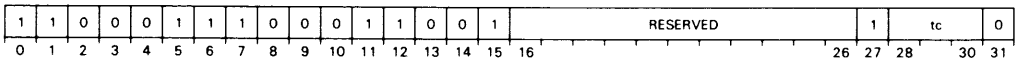
Upon completion, the contents of FPAC1, FPAC2, and FPAC3 are undefined.

> **NOTE:** *If the input value in FPAC0 is less than 0, then the processor sets the INV bit in the FPSR to one and returns error code two to the INP bits of the FPSR.*

This instruction is one of the optional floating-point intrinsic instructions (IIS).

Wide Forward Search Queue and Skip

WFStc



The WFStc instruction searches forward through a 32-bit data field. If the search is successful, it skips the next two single-word instructions.

AC1 and AC3 identify a location in a data field as a beginning data element in the queue search. AC1 contains an effective address. AC3 contains a double word offset (a signed integer). The processor locates the beginning data element by calculating the effective address and adding the word offset.

The double word at the top of the wide stack contains the mask word. The bits in the mask word identify the test location bits to sample.

Bits 28-30 of the WFStc instruction specify the search condition.

tc Value	Bits 28-30 Encoding *	Meaning
SS	0 0 0	Some of the sampled test location bits are 1
SC	0 0 1	Some of the sampled test location bits are 0
AS	0 1 0	All of the sampled test location bits are 1
AC	0 1 1	All of the sampled test location bits are 0
E	1 0 0	The mask and test location are equal
GE	1 0 1	The mask is greater than or equal to the test locaiton
LE	1 1 0	The mask is less than or equal to the test location
NE	1 1 1	The mask and test location are not equal

*The instruction treats the values contained in the mask and in the test location as unsigned integers for the E, GE, LE, and NE test conditions.

The search begins with the addressed data element, and compares it with the 32-bit mask word. The search continues until the processor reaches either the tail of the queue or the data element that meets the test condition.

If the search is successful, AC1 contains the effective address of the data element, and the processor then skips the next two single-word insructions. The processor will not honor interrupts between the time that it completes a successful search and executes the PC+4 instruction.

> **NOTE:** *After a successful search terminates, a new beginning pointer must be placed in AC1 if the search is to be continued through the remainder of the queue.*

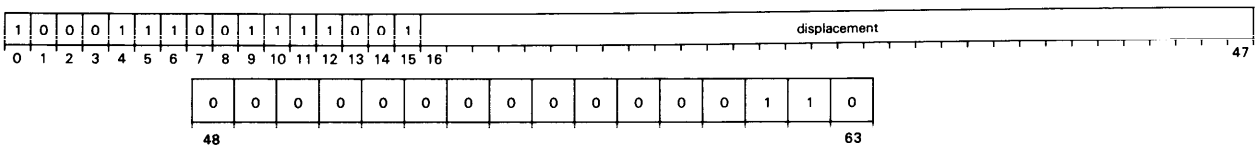
If the search fails, AC1 contains the effective address of the last data element searched, and the processor then executes the next instruction. The processor will honor interrupts between the time that it completes an unsuccessful search and executes the PC+2 instruction.

If the processor interrupts the search, AC1 contains the effective address of the next data element to examine, and the processor then skips the next instruction. The processor will honor interrupts between the time that the interrupt occurs and the processor executes the PC+3 instruction.

For all returns, the contents of CARRY, OVR, WSP, AC0, AC2, and AC3 remain unchanged.

Floating-Point Tangent Double

WFTAND displacement



Computes the tangent of the 64-bit floating-point value (in radians) in FPAC0, and places the result in FPAC0.

NOTE: *If hardware support is presently unavailable, the processor performs an LPSHJ instruction function and computes the effective address, E (using the displacement as a non-indirectable PC-relative offset). Uses E as the address of a routine in a run-time library which performs the tangent function. Return from this software emulator is by the WPOPJ instruction. The effective address must be in the current ring.*

Returns the floating-point result to FPAC0. Updates the Z and N flags of the floating-point status register to reflect the new contents of FPAC0.

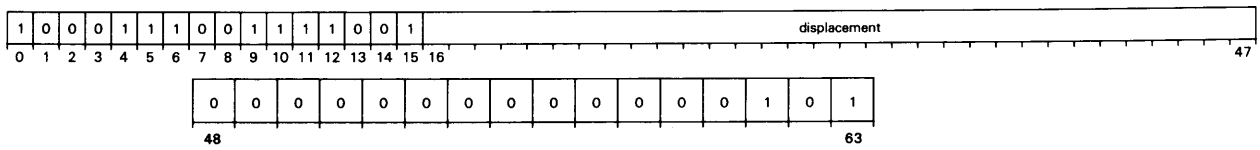
Upon completion, the contents of FPAC1, FPAC2, and FPAC3 are undefined.

NOTE: *If the input value in FPAC0 will produce a result which overflows (i.e, integer multiples of values near PI/2), the processor sets the INV bit in the FPSR to one and returns error code six to the INP bits of the FPSR.*

This instruction is one of the optional floating-point intrinsic instructions (IIS).

Floating-Point Tangent Single

WFTANS displacement



Computes the tangent of the 32-bit floating-point value (in radians) in FPAC0, and places the result in FPAC0.

NOTE: *If hardware support is presently unavailable, the processor performs an LPSHJ instruction function and computes the effective address, E (using the displacement as a non-indirectable PC-relative offset). Uses E as the address of a routine in a run-time library which performs the tangent function. Return from this software emulator is by the WPOPJ instruction. The effective address must be in the current ring.*

Returns the floating-point result to FPAC0 bits 0-31 and sets bits 32-63 to 0. Updates the Z and N flags of the floating-point status register to reflect the new contents of FPAC0.

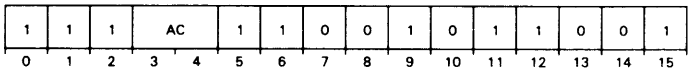
Upon completion, the contents of FPAC1, FPAC2, and FPAC3 are undefined.

NOTE: *If the input value in FPAC0 will produce a result which overflows (i.e, integer multiples of values near PI/2), the processor sets the INV bit in the FPSR to one and returns error code six to the INP bits of the FPSR.*

This instruction is one of the optional floating-point intrinsic instructions (IIS).

Wide Halve

WHLV *ac*



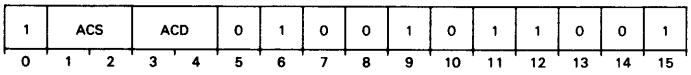
Divides the 32-bit contents of the specified accumulator by two and rounds the result toward zero.

The signed, 32-bit two's complement number contained in the specified accumulator is divided by two and rounded toward zero. The result is placed in the specified accumulator.

This instruction leaves carry unchanged; *overflow* is 0.

Wide Increment

WINC *acs,acd*

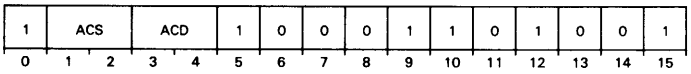


Increments an integer contained in an accumulator.

The instruction increments the 32-bit contents of ACS by 1 and loads the result into ACD. Sets carry to the value of ALU carry. Sets *overflow* to 1, if there is an ALU overflow. The contents of ACS remain unchanged, unless ACS equals ACD.

Wide Inclusive OR

WIOR *acs,acd*

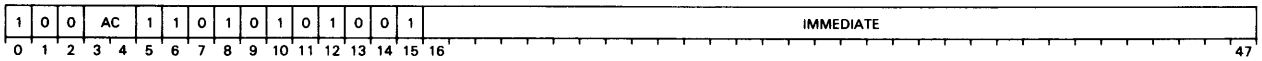


Performs an inclusive OR between two accumulators.

Forms the logical inclusive OR between corresponding bits of ACS and ACD. Loads the 32-bit result into ACD. The contents of ACS remain unchanged. Carry is unchanged and *overflow* is 0.

Wide Inclusive OR Immediate

WIORI *i,ac*

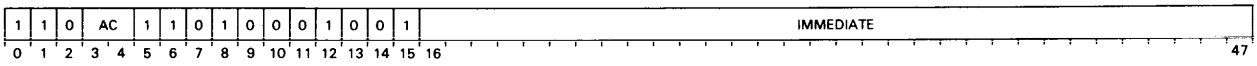


Performs an inclusive OR between the contents of the immediate field and an accumulator.

The instruction forms the logical inclusive OR between corresponding bits of the specified accumulator and the value contained in the literal field. The instruction places the result of the inclusive OR in the specified accumulator. Carry is unchanged and *overflow* is 0.

Wide Load with Wide Immediate

WLDAI *i,ac*

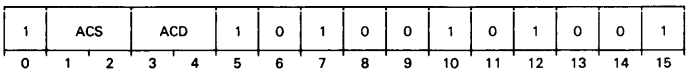


Loads an accumulator with the contents of an immediate value.

Loads the 32-bit value contained in the immediate field into the specified accumulator. Carry is unchanged and *overflow* is 0.

Wide Load Byte

WLDB *acs,acd*

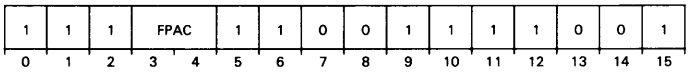


Loads a byte in memory into an accumulator.

Uses the 32-bit byte address contained in ACS to load a byte into ACD. Sets bits 0–23 of ACD to zero. Bits 24–31 of ACD contain a copy of the contents of the addressed byte. The contents of ACS remain unchanged, unless ACS and ACD are the same accumulator. Carry is unchanged and *overflow* is 0.

Wide Load Integer

WLDI *fpac*



Translates up to a 16-digit decimal integer from memory to floating-point format and places the result in a floating-point accumulator.

AC1 must contain the data-type indicator describing the integer.

AC3 must contain a 32-bit byte pointer pointing to the high-order byte of the integer in memory.

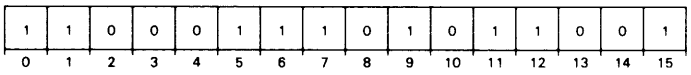
Uses AC1 and AC3 to convert a decimal integer to floating-point form. Normalizes the result and places it in the specified FPAC. Updates the *Z* and *N* flags in the FPSR to describe the new contents of the specified FPAC. Leaves the decimal number unchanged in memory.

By convention, the first byte of a number stored according to data type 7 contains the sign and exponent of the floating-point number. The instruction copies each byte (following the lead byte) directly to the mantissa of the specified FPAC. It then sets to zero each low-order byte in the FPAC that does not receive data from memory.

Upon successful completion, AC0 and AC1 remain unchanged. AC2 contains the original contents of AC3. AC3 points to the first byte following the integer field. Carry is unchanged and *overflow* is 0.

Wide Load Integer Extended

WLDIX



Distributes a decimal integer of data type 0, 1, 2, 3, 4, or 5 into the four FPACs.

AC1 must contain the data-type indicator describing the integer.

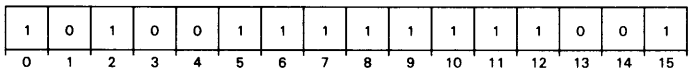
AC3 must contain a 32-bit byte pointer which is the address of the high-order byte of the integer.

The instruction uses the contents of AC3 to reference the integer. Extends the integer with high-order zeros until it is 32 digits long. Divides the integer into four units of 8 digits each and converts each unit to a floating-point number. Places the number obtained from the 8 high-order digits into FAC0. Places the number obtained from the next 8 digits into FAC1. Places the number obtained from the next 8 digits into FAC2. Places the number obtained from the low-order 8 bits into FAC3. Sets the sign of each FPAC by checking the number just loaded into the FPAC. If the FPAC contains a nonzero number, then sets the sign of the FPAC to be the sign of the integer. If the FPAC contains an 8-digit zero, sets the FPAC to true zero. The *Z* and *N* flags in the floating-point status register are unpredictable.

Upon successful termination, the contents of AC0 and AC1 remain unchanged. AC2 contains the original contents of AC3. AC3 points to the first byte following the integer field. Carry is unchanged and *overflow* is 0.

Wide Load Map

WLMP



Loads a series of double words into successive map registers.

This command loads the contents of the double word specified by AC2 into the map slot specified by AC0. It decrements the count in AC1 by one, increments the map slot number in AC0 by one, and increments the address in AC2 by two; this continues until AC1 contains zeros in bits 16–31. Upon completion, AC0 references the map slot following the last slot loaded; AC1 contains a zero in bits 16–31; AC2 contains the address of the word following the last double word loaded; AC3 and carry remain unchanged; *overflow* is 0.

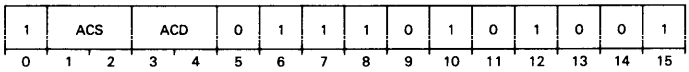
If bits 16–31 of AC1 all initially contain zeros, the instruction performs no operation.

NOTE: *This is a privileged instruction.*

Refer to the appropriate functional characteristics manual for more information.

Wide Locate Lead Bit

WLOB *acs,acd*

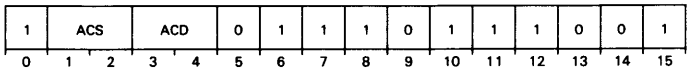


Counts the number of high-order zeroes in ACS.

The instruction counts the high-order zeroes in ACS. Adds the count of high-order zeroes to the 32-bit, signed contents of ACD. Stores the result of the add in ACD. The contents of ACS remain unchanged, unless ACS and ACD are the same accumulator. Carry is unchanged and *overflow* is 0.

Wide Locate and Reset Lead Bit

WLRB *acs,acd*



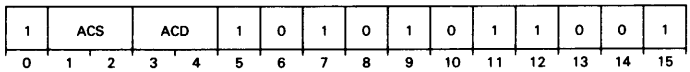
Counts the number of high-order zeroes in ACS.

The instruction counts the high-order zeroes in ACS. Adds the count of high-order zeroes to the 32-bit, signed contents of ACD. Stores the result in ACD. Sets the leading bit of ACS to 0. Carry is unchanged and *overflow* is 0.

If ACS equals ACD, then sets the leading bit to 0 and adds nothing to the contents of the specified accumulator.

Wide Logical Shift

WLSH *acs,acd*



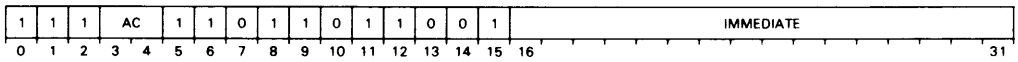
Shifts the 32-bit contents of ACD either left or right.

Bits 24–31 of ACS specify the number of bits to shift ACD. If this number is positive, then the instruction shifts the contents of ACD the appropriate number of bits to the left. If this number is negative, then the instruction shifts the contents of ACD the appropriate number of bits to the right. If ACS contains zero, then no shifting occurs. The instruction ignores bits 0–23 of ACS.

Bits shifted out during this instruction are lost. Zeroes fill the vacated bit positions. The contents of ACS remain unchanged, unless ACD equals ACS. Carry is unchanged and *overflow* is 0.

Wide Logical Shift With Narrow Immediate

WLSHI *i,ac*



Shifts the 32-bit contents of an accumulator either left or right.

Bits 24–31 specify the number of bits to shift AC. If this number is positive (1 to 32_{10}), then the instruction shifts the contents of AC the appropriate number of bits to the left. If this number is negative (-1 to -32_{10}), then the instruction shifts the contents of AC the appropriate number of bits to the right. If the immediate contains zero, then no shifting occurs. Bits 16–23 of the immediate field must be identical to bit 24; otherwise, results are indeterminate. The processor sign extended the narrow immediate field to 32 bits.

Bits shifted out during this instruction are lost. Zeroes fill the vacated bit positions.

Carry is unchanged and *overflow* is 0.

Wide Logical Shift Immediate

WLSI *n,ac*

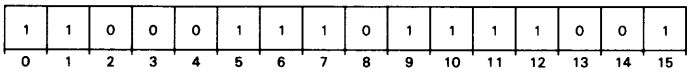
Shifts the contents of an accumulator left, as indicated by an immediate value.

Shifts the contents of the specified accumulator to the left $n+1$ positions, where n is in the range of 0 to 3. Carry is unchanged and *overflow* is 0.

NOTE: *The assembler takes the coded value of* n *and subtracts one from it before placing it in the immediate field. Therefore, the programmer should code the exact value to be shifted.*

Wide Load Sign

WLSN



Evaluates a decimal number as zero or nonzero and the sign as positive or negative.

AC1 must contain the data type indicator describing the number.

AC3 must contain a byte pointer which is the address of the high-order byte of the number.

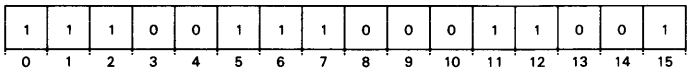
The instruction evaluates a decimal number in memory and returns in AC1 a code that classifies the number as zero or nonzero and identifies its sign. The meaning of the returned code is as follows:

Value of Number	Code
Postive nonzero	+1
Negative nonzero	-1
Positive zero	0
Negative zero	-2

Upon successful termination, the contents of AC0 remain unchanged; AC1 contains the value code; AC2 contains the original contents of AC3; and the contents of AC3 are unpredictable. The contents of the addressed memory locations remain unchanged. Carry is unchanged and *overflow* is 0.

Wide Mask, Skip and Store if Equal

WMESS
unsuccessful exit
successful exit



The **WMESS** instruction tests and sets multiple bits of a double word in memory. The instruction reads the double word addressed by AC2 and then performs an exclusive OR of the double word with the contents of AC0.

NOTE: *The operation is guaranteed indivisible if the double word addressed by AC2 is double-word aligned.*

If all of the resultant bits that equal 1 from the XOR also correspond to the bits that equal 0 in AC3, then the comparison is successful. The processor exchanges the values in AC1 with the double word in memory. Then it skips the next instruction.

If any bit resulting from XOR equals 1, and if it also corresponds to a bit that equals 1 in AC3, then the comparison is unsuccessful. The processor loads the value from the double word in memory into AC1 and executes the next instruction.

Carry is unchanged and *overflow* is 0.

AC0 contains 32 bits that the processor compares (exclusive OR) with the 32 bits in memory. Upon completion, AC0 remains unchanged.

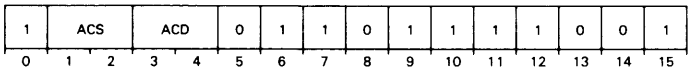
AC1 contains 32 bits that the processor exchanges with the 32 bits in memory when the result of the comparison is true. Upon completion, AC1 always contains the initial 32 bits of the double word addressed by AC2. (The processor pushes a fault return block and updates AC1 with an error code if a protection fault occurs -- an invalid read or write memory reference.)

AC2 contains the address of the data element to test. The instruction does not permit indirect addressing. Upon completion, AC2 remains unchanged.

AC3 contains 32 bits that the processor compares (logical AND) with the results of the exclusive OR operation. Upon completion, AC3 remains unchanged.

Wide Move

WMOV *acs,acd*



Moves a copy of the 32-bit contents of ACS into ACD.

The contents of ACS remain unchanged. Carry is unchanged and *overflow* is 0.

Wide Move Right

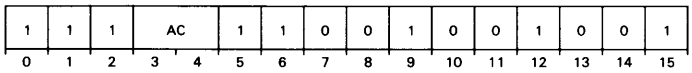
WMOVR *ac*

Converts a byte pointer to a word pointer.

Moves the contents of an AC right one bit, shifting in a zero to bit 0. Carry is unchanged and *overflow* is 0.

Wide Modify Stack Pointer

WMSP *ac*



Changes the value of the stack pointer and tests for potential overflow/underflow.

The contents of AC specify the number of double words to add to the WSP. The instruction shifts the contents of the specified accumulator left one bit and temporarily saves the result. Adds the shifted value to the contents of the WSP and temporarily saves the result. Checks for fixed-point overflow resulting from the shift and addition. If overflow occurs, the processor does not alter WSP and treats the overflow as a stack fault. AC1 contains the code 1.

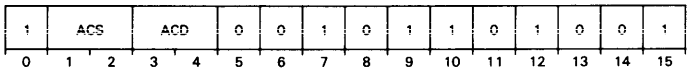
If no overflow occurs, the instruction checks the value of the accumulator. If the value is positive, the processor checks the temporary WSP against the stack limit for stack overflow; if negative, against the stack base for stack underflow. If underflow or overflow does not occur, the instruction places the temporary WSP into the contents of the WSP.

If either overflow or underflow occurs, the instruction does not alter WSP and a stack fault occurs. When a stack underflow occurs, the WMSP instruction uses the stack pointer (not the stack limit) to push the fault block. In the fault block, AC1 contains the code 1. The PC in the return block points to this instruction.

This instruction does not change carry; *overflow* is 0.

Wide Multiply

WMUL *acs,acd*



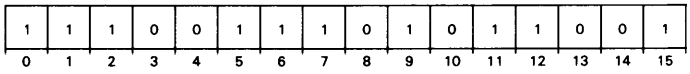
Multiplies two integers contained in accumulators.

The instruction multiplies the 32-bit, signed integer contained in ACD by the 32-bit, signed integer contained in ACS. Places the 32 least significant bits of the result in ACD. The contents of ACS and carry remain unchanged.

If the result is outside the range of −2,147,483,648 to +2,147,483,647 inclusive, sets *overflow* to 1; otherwise, *overflow* is 0. ACD will contain the 32 least significant bits of the result.

Wide Signed Multiply

WMULS

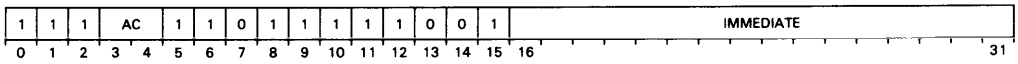


Multiplies two integers contained in accumulators.

The instruction multiplies the 32-bit, signed integer contained in AC1 by the 32-bit, signed integer contained in AC2. Adds the 32-bit signed integer contained in AC0 to the 64-bit result. Loads the 64-bit result into AC0 and AC1. AC0 contains the 32 high-order bits. AC2 and carry remain unchanged. *Overflow* is 0.

Wide Add with Narrow Immediate

WNADI *i,ac*

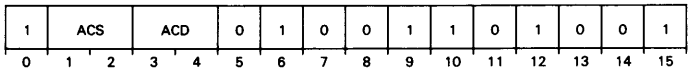


Adds an immediate value to an integer contained in an accumulator.

The instruction sign extends the two's complement literal value contained in the immediate field to 32 bits. Adds the sign extended value to the 32-bit integer contained in the specified accumulator. Loads the result into the specified accumulator. Sets carry to the value of ALU carry. Sets *overflow* to 1, if there is an ALU overflow.

Wide Negate

WNEG *acs,acd*



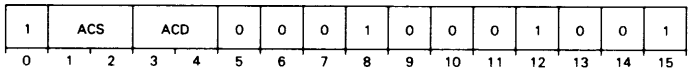
Negates the contents of an accumulator.

The instruction forms the two's complement of the 32-bit contents of ACS. Loads the result into ACD. Sets carry to the value of ALU carry. Sets *overflow* to 1, if there is an ALU overflow. The contents of ACS remain unchanged, unless ACS equals ACD.

NOTE: *Negating the largest negative 32-bit integer (10000000000_8) sets overflow to 1.*

Wide Pop Accumulators

WPOP *acs,acd*



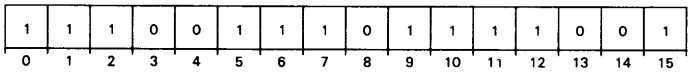
Pops up to four double words off the top of the wide stack and places them in the specified 32-bit accumulators.

Pops the top double word off the wide stack and places it in ACS. Pops the next double word off the wide stack and places it in ACS-1, and so on, until all specified accumulators have been loaded. If necessary, the accumulators wrap around, with AC3 following AC0, until all specified accumulators have been loaded. If ACS equals ACD, then the instruction pops only one double word off of the wide stack and places it in the specified accumulator.

The instruction decrements the contents of WSP by twice the number of double words popped, then checks for stack underflow. Carry is unchanged and *overflow* is 0.

Wide Pop Block

WPOPB



Returns control from an intermediate-level interrupt, from an extended operation (WXOP), or from a breakpoint (BKPT) instruction (when removing the BKPT instruction before returning from the breakpoint handler).

Pops six double words off the wide stack and places them in the appropriate locations. The popped words and their destinations are as follows:

Double Word Popped	Destination
1	Bit 0 to carry; bits 1-31 to PC
2	AC3
3	AC2
4	AC1
5	AC0
6	Bits 0–15 are the PSR; bit 16 is 0; bits 17-31 are the frame size (double words).

If the instruction specifies an inward ring crossing, then a protection fault occurs and the current wide stack remains unchanged. Note that the return block pushed as a result of the protection fault will contain undefined information. After the fault return block is pushed, AC0 contains the contents of the PC (which point to the instruction that caused the fault) and AC1 contains the code 8.

If the instruction specifies an intra-ring address, it pops the six-double-word block and saved frame area, then checks for stack underflow. If underflow has occurred, a stack underflow fault results. Note that the return block pushed as a result of the stack underflow will contain undefined information. After the fault return block is pushed, AC0 contains the contents of the PC (which point to the instruction that caused the fault) and AC1 contains the code 3. If there is no underflow, execution continues with the location addressed by the program counter.

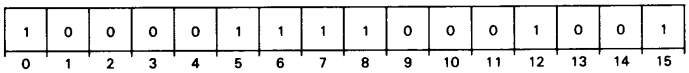
If the instruction specifies an outward ring crossing, it pops the six-double-word return block and saved frame area and checks for stack underflow. If underflow has occurred, a stack underflow fault results. Note that the return block pushed as a result of the stack underflow will contain undefined information. After the fault return block is pushed, AC0 contains the contents of the PC (which point to the instruction that caused the fault) and AC1 contains the code 3. If there is no underflow, the instruction stores WSP and WFP in the appropriate page zero locations of the current segment. It then performs the outward ring crossing and loads the wide stack registers with the contents of the appropriate page zero locations of the new ring. Loads WSP with the value:

(*current contents of WSP*) − (2 x (*frame size*))

If *frame size* is greater than 0, then the processor checks for stack underflow. If underflow has occurred, a stack underflow fault results. Note that the return block pushed as a result of the stack underflow will contain undefined information. After the fault return block is pushed, AC0 contains the contents of the PC (which point to the instruction that caused the fault) and AC1 contains the code 3. If there is no underflow, execution continues with the location addressed by the program counter.

Wide Pop PC and Jump

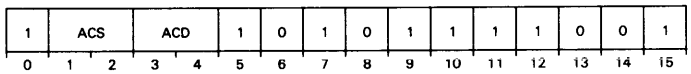
WPOPJ



Pops the top 28-bit value off the wide stack, loads it into the PC, then checks for stack underflow. Carry is unchanged and *overflow* is 0.

Wide Push Accumulators

WPSH *acs,acd*



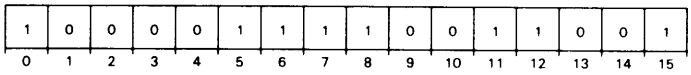
Pushes the contents of the specified 32-bit accumulators onto the top of the wide stack.

Pushes the contents of ACS onto the top of the wide stack, then pushes the contents of next sequential accumulators up to and including ACD. If necessary, the accumulators wrap around, with AC0 following AC3, until the contents of all specified accumulators have been pushed. If ACS equals ACD, then the instruction pushes the contents of only one accumulator onto the wide stack.

The instruction increments the contents of WSP by two times the number of accumulators pushed (32-bit accumulators) then checks for stack overflow. Carry is unchanged and *overflow* is 0.

Wide Restore

WRSTR



Returns control from a base-level interrupt.

When this instruction is used, the wide stack should contain the following information, in the given order:

Doubleword	Contents	Size of Word	Notes
11	**WFP**	32 bits	
10	**WSP**	32 bits	
9	**WSL**	32 bits	
8	**WSB**	32 bits	
7	**0,SFA**	32 bits	Stack fault address (bits 1-15)
6	**PSR,0**	32 bits	
5	**AC0**	32 bits	
4	**AC1**	32 bits	
3	**AC2**	32 bits	
2	**AC3**	32 bits	
1	**Carry, PC**	32 bits	This is the top of the wide stack.

The instruction checks to see if the ring crossing specified is inward. If the crossing is inward, a protection fault occurs (code=8 in AC1).

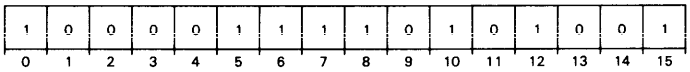
If the crossing is not inward, the instruction pops the return block on top of the wide stack and places the block contents in the appropriate registers. Next, the instruction pops the stack registers and the stack fault address, temporarily saves them, and checks for stack underflow. If underflow has occurred, a stack underflow fault results (code=3 in AC1). Note that the fault uses the original stack parameters, not the new ones. If no underflow occurs, further actions depend upon the type of ring call.

If the restore is to be to the same ring, the instruction places the temporarily saved stack management information in the four stack registers. Stores the stack fault address in the stack fault pointer of the current segment. Execution continues with the location specified by the PC.

If the ring crossing is outward, the instruction temporarily stores the stack management information internally into the appropriate page zero locations of the current segment. Performs the outward ring crossing. Loads the stack registers with the contents of the appropriate page zero locations of the new segment. If the argument count (from call) is greater than zero, the instruction checks for stack underflow. If underflow has occurred, a stack underflow fault results (code=3 in AC1). If underflow has not occurred, execution continues with the location specified by the PC.

Wide Return

WRTN

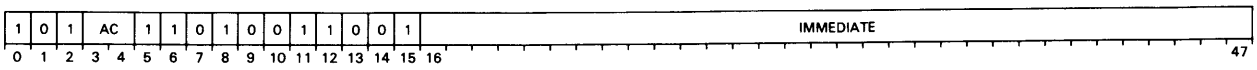


Returns control from a subroutine.

Returns control from subroutines that issue a **WSAVS**, **WSAVR**, **WSSVS**, or a **WSSVR** instruction at their entry point. Places the contents of WFP in WSP and executes a **WPOPB** instruction. Places the popped value of AC3 in WFP.

Wide Skip on All Bits Set in Accumulator

WSALA *i,ac*

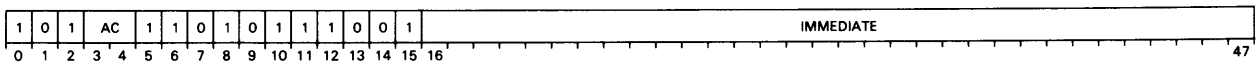


Logically ANDs an immediate value with the contents of an accumulator and skips depending on the result of the AND.

Performs a logical AND on the contents of the immediate field and the complement of the contents of the specified accumulator. If the result of the AND is zero, then execution skips the next sequential word before continuing. If the result of the AND is nonzero, then execution continues with the next sequential word. The contents of the specified accumulator remain unchanged. Carry is unchanged and *overflow* is 0.

Wide Skip on All Bits Set in Double-Word Memory Location

WSALM *i,ac*



Logically ANDs an immediate value with the complement of a memory word and skips depending on the result of the AND.

Performs a logical AND on the contents of the immediate field and the complement of the double word addressed by the specified accumulator. If the result of the AND is zero, then execution skips the next sequential word before continuing. If the result of the AND is nonzero, then execution continues with the next sequential word. The contents of the specified accumulator and memory location remain unchanged. Carry is unchanged and *overflow* is 0.

Wide Skip on Any Bit Set in Accumulator

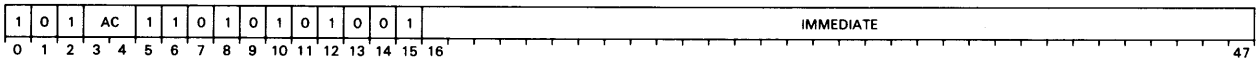
WSANA *i,ac*

Logically ANDs an immediate value with the contents of an accumulator and skips depending on the result of the AND.

Performs a logical AND on the contents of the immediate field and the contents of the specified accumulator. If the result of the AND is nonzero, then execution skips the next sequential word before continuing. If the result of the AND is zero, then execution continues with the next sequential word. The contents of the specified accumulator remain unchanged. Carry is unchanged and *overflow* is 0.

Wide Skip on Any Bit Set in Double-Word Memory Location

WSANM *i,ac*

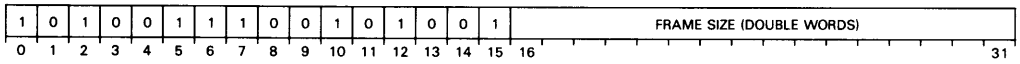


Logically ANDs an immediate value with the contents of a memory word and skips depending on the result of the AND.

Performs a logical AND on the contents of the immediate field and the contents of the double word addressed by the specified accumulator. If the result of the AND is nonzero, then execution skips the next sequential word before continuing. If the result of the AND is zero, then execution continues with the next sequential word. The contents of the specified accumulator and memory location remain unchanged. Carry is unchanged and *overflow* is 0.

Wide Save/Reset Overflow Mask

WSAVR *frame size*

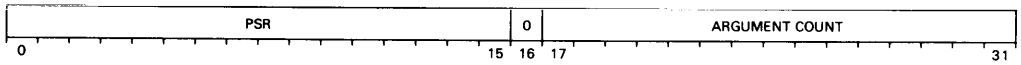


Pushes a partial return block onto the wide stack, resets *OVK*, and increments the wide stack pointer by *frame size*.

The instruction checks for stack overflow. If an overflow would occur, then control transfers to the wide stack fault routine. If no overflow would occur, then the instruction pushes five double words of a wide six-double-word return block onto the wide stack. The words pushed have the following contents:

Double Word Pushed	Contents
1	AC0
2	AC1
3	AC2
4	Previous WFP
5	Bit 0 equal CARRY Bits 1–31 equals AC3 bits 1–31 (or return PC value for XCALL and LCALL)

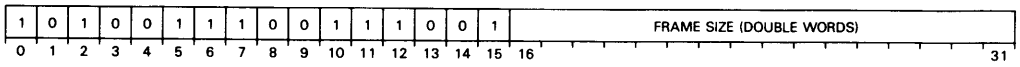
Note that the five words described above do not make up the entire return block. Either the **LCALL** or the **XCALL** instruction pushes the first double word of the return block onto the wide stack. This word has the following format:



After pushing the return block, the instruction places the value of the stack pointer in WFP and AC3. Multiplies the 16-bit, unsigned integer contained in the frame size by 2. Adds the result to WSP, which reserves the space for local variables. Sets *OVK* to 0, disabling integer overflow.

Wide Save/Set Overflow Mask

WSAVS *frame size*

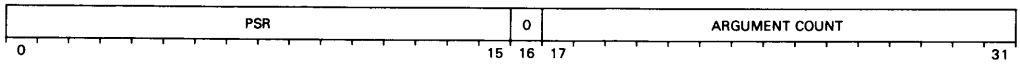


Pushes a partial return block onto the wide stack, resets WSP and WFP, sets *OVK* to 1, and increments the wide stack pointer by *frame size.*

The instruction checks for stack overflow. If an overflow would occur, then control transfers to the wide stack fault routine. If no overflow would occur, then the instruction pushes five double words of a wide six-double-word return block onto the stack. The words pushed have the following contents.

Double Word Pushed	Contents
1	AC0
2	AC1
3	AC2
4	Previous WFP
5	Bits 0 equals CARRY Bits 1–31 equal AC3 bits 1–31 (or return PC value for XCALL and LCALL).

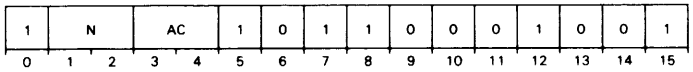
Note that the five double words described above do not make up the entire return block. Either the **LCALL** or the **XCALL** instruction pushes the first double word of the return block onto the wide stack. This word has the following format:



After pushing the return block, the instruction places the value of WSP in WFP and AC3. Multiplies the 16-bit, unsigned integer contained in the frame size by 2. Adds the result to WSP, which reserves the space for local variables. Sets *OVK* to 1, enabling integer overflow.

Wide Subtract Immediate

WSBI *n,ac*



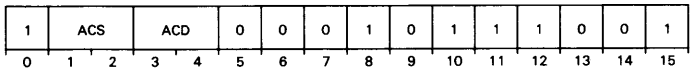
Subtracts an integer in the range 1 to 4 from an integer contained in an accumulator.

The instruction subtracts the value $n+1$ from the value contained in the specified accumulator. Stores the result in the specified accumulator. Sets carry to the value of ALU carry. Sets *overflow* to 1, if there is an ALU overflow.

> **NOTE:** *The assembler takes the coded value of* n *and subtracts 1 from it before placing it in the immediate field. Therefore, the programmer should code the exact value to be subtracted.*

Wide Skip if Equal to

WSEQ *acs,acd*



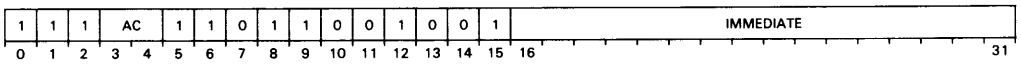
Compares one integer to another and skips if the two integers are equal.

The instruction compares the 32-bit integer contained in ACS to the 32-bit integer in ACD. If the integer contained in ACS is equal to the integer contained in ACD, the next 16-bit word is skipped; otherwise, the next word is executed.

If ACS and ACD are the same accumulator, then the instruction compares the integer contained in the accumulator to 0. The skip will occur if the integer equals 0. Carry is unchanged and *overflow* is 0.

Wide Skip if AC Equal to Immediate

WSEQI *i,ac*



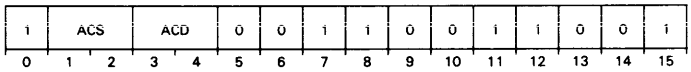
Compares one integer to another and skips if the first is equal to the second.

Sign extends the 16-bit immediate field. Compares this 32-bit number to the contents of the AC. If the contents of the AC are equal to the contents of the immediate, then the next sequential word is skipped; otherwise, the next word is executed.

Contents of AC and carry remain unchanged. *Overflow* is 0.

Wide Signed Skip if Greater than or Equal to

WSGE *acs,acd*



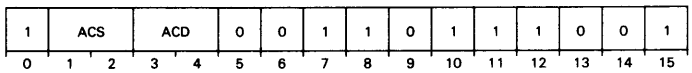
Compares one integer to another and skips if the first is greater than or equal to the second.

The instruction compares the signed, 32-bit integer contained in ACS to the signed, 32-bit integer in ACD. If the integer contained in ACS is greater than or equal to the integer contained in ACD, then the next word is skipped; otherwise, the next instruction is executed.

If ACS and ACD are the same accumulator, then the instruction compares the integer contained in the accumulator to 0. The skip will occur if the integer is greater than or equal to 0. Carry is unchanged and *overflow* is 0.

Wide Signed Skip if Greater than

WSGT *acs,acd*



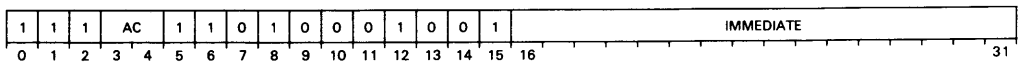
Compares one integer to another and skips if the first is greater than the second.

The instruction compares the signed, 32-bit integer contained in ACS to the signed 32-bit integer in ACD. If the integer contained in ACS is greater than the integer contained in ACD, the next word is skipped; otherwise, the next word is executed.

If ACS and ACD are the same accumulator, then the instruction compares the integer contained in the accumulator to 0. The skip will occur if the integer is greater than 0. Carry is unchanged and *overflow* is 0.

Wide Skip if AC Greater than Immediate

WSGTI *i,ac*



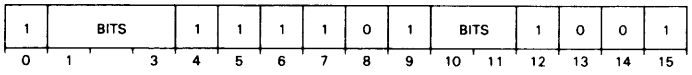
Compares one integer to another and skips if the first is greater than the second.

Sign extends the 16-bit immediate field. Compares this 32-bit number to the contents of the AC. If the contents of the AC are greater than the contents of the immediate, then the next sequential word is skipped; otherwise, the next word is executed.

Contents of AC and carry remain unchanged. *Overflow* is 0.

Wide Skip on Bit Set to One

WSKBO *bit number*

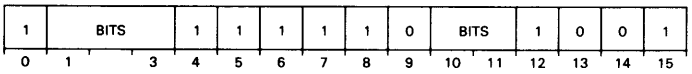


Tests a specified bit in AC0 and skips if the bit is 1.

The instruction uses the bits specified in bits 1–3 and 10–11 to specify a bit position in the range 0–31. This number specifies one bit in AC0; the value 0 specifies the highest-order bit and the value 31 specifies the lowest-order bit. If the specified bit has the value 1, then the next sequential word is skipped. If the bit has the value 0, then the next sequential word is executed. The contents of AC0 remain unchanged. Carry is unchanged and *overflow* is 0.

Wide Skip on Bit Set to Zero

WSKBZ *bit number*

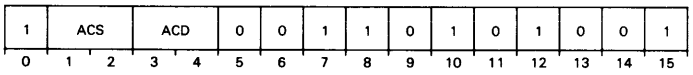


Tests a specified bit in AC0 and skips if the bit is 0.

The instruction uses the bits specified in bits 1–3 and 10–11 to specify a bit position in the range 0–31. This number specifies one bit in AC0; the value 0 specifies the highest-order bit and the value 31 specifies the lowest-order bit. If the specified bit has the value 0, then the next sequential word is skipped. If the bit has the value 1, then the next sequential word is executed. The contents of AC0 remain unchanged. Carry is unchanged and *overflow* is 0.

Wide Signed Skip if Less than or Equal to

WSLE *acs,acd*



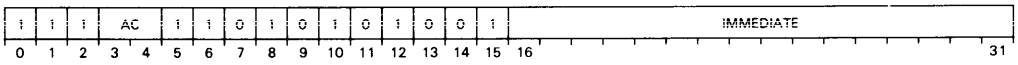
Compares one integer to another and skips if the first is less than or equal to the second.

The instruction compares the signed, 32-bit integer contained in ACS to the signed, 32-bit integer in ACD. If the integer contained in ACS is less than or equal to the integer contained in ACD, the next word is skipped; otherwise, the next sequential word is executed.

If ACS and ACD are the same accumulator, then the instruction compares the integer contained in the accumulator to 0. The skip will occur if the integer is less than or equal to 0. Carry is unchanged and *overflow* is 0.

Wide Skip if AC Less than or Equal to Immediate

WSLEI *i,ac*



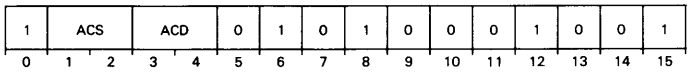
Compares one integer to another and skips if the first is less than or equal to the second.

Sign extends the 16-bit immediate field. Compares this 32-bit number to the contents of the AC. If the contents of the AC are less than or equal to the contents of the immediate, then the next sequential word is skipped; otherwise, the next word is executed.

Contents of AC and carry remain unchanged. *Overflow* is 0.

Wide Signed Skip if Less than

WSLT *acs,acd*



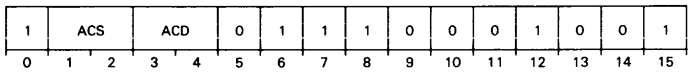
Compares one integer to another and skips if the first is less than the second.

The instruction compares the signed, 32-bit integer contained in ACS to the signed, 32-bit integer in ACD. If the integer contained in ACS is less than the integer contained in ACD, the next word is skipped; otherwise, the next sequential word is executed.

If ACS and ACD are the same accumulator, then the instruction compares the integer contained in the accumulator to 0. The skip will occur if the integer is less than 0. Carry is unchanged and *overflow* is 0.

Wide Skip on Nonzero Bit

WSNB *acs,acd*



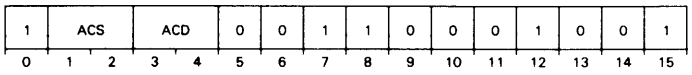
Tests the value of an addressed bit and skips if the bit is one.

The instruction forms a bit pointer from the contents of ACS and ACD. ACS contains the high-order bits of the bit pointer; ACD contains the low-order bits. ACS and ACD can be specified to be the same accumulator; in this case, the specified accumulator supplies the low-order bits of the bit pointer. The high-order bits are treated as if they were zero in the current segment.

The instruction checks the value of the bit referenced by the bit pointer. If the bit has the value 1, the next sequential word is skipped. If the bit has the value 0, the next sequential instruction is executed. Carry is unchanged and *overflow* is 0.

Wide Skip if Not Equal to

WSNE *acs,acd*



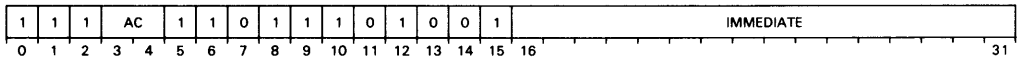
Compares one integer to another and skips if the two are not equal.

The instruction compares the 32-bit integer contained in ACS to the 32-bit integer in ACD. If the integer contained in ACS is not equal to the integer contained in ACD, then execution skips the next word; otherwise, execution proceeds with the next sequential word.

If ACS and ACD are the same accumulator, then the instruction compares the integer contained in the accumulator to 0. The skip will occur if the integer does not equal 0. Carry is unchanged and *overflow* is 0.

Wide Skip if AC Not Equal to Immediate

WSNEI *i,ac*



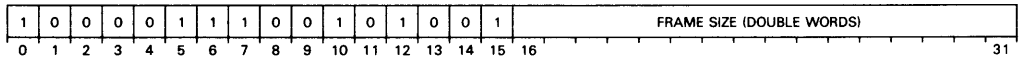
Compares one integer to another and skips if the first is not equal to the second.

Sign extends the 16-bit immediate field. Compares this 32-bit number to the contents of the AC. If the contents of the AC are not equal to the contents of the immediate, then the next sequential word is skipped; otherwise, the next word is executed.

Contents of AC and carry remain unchanged. *Overflow* is 0.

Wide Special Save/Reset Overflow Mask

WSSVR *frame size*



Pushes a wide return block onto the wide stack, sets *OVK* to 0, and increments the stack pointer by *frame size*.

The instruction checks for stack overflow. If executing the instruction would cause an overflow, the instruction transfers control to the wide stack fault handler. The PC in the fault return block will contain the address of the **WSSVR** instruction.

Pushes a wide return block onto the wide stack. After pushing the sixth double word, places the value of WSP in WFP and AC3. Increments WSP by twice the frame size to reserve the space for local variables. Sets *OVK* to 0, which disables integer overflow. Sets *OVR* to 0.

The structure of the wide return block pushed is as follows:

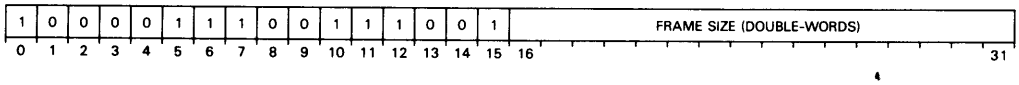
Double Word in Block	Contents
1	PSR, and zeroes in the unused bits
2	AC0
3	AC1
4	AC2
5	Previous WFP
6	Carry and AC3 1–31 (or return PC)

NOTE: *This instruction saves the information required by the* **WRTN** *instruction.*

This instruction is typically executed after an **XJSR** or **LJSR** instruction. Note that neither of these jump instructions can perform a cross ring call. However, they may be used with **WSSVS** to perform an intra-ring transfer to a subroutine that requires no parameters and that uses **WRTN** to return control back to the calling sequence.

Wide Special Save/Set Overflow Mask

WSSVS *frame size*



Pushes a wide return block onto the wide stack, sets *OVK* to 1, and increments the wide stack pointer by *frame size.*

The instruction checks for stack overflow. If executing the instruction would cause an overflow, the instruction transfers control to the wide stack fault handler. The PC in the fault return block will contain the address of the **WSSVS** instruction.

If no overflow would occur, the instruction pushes a wide return block onto the wide stack. After pushing the sixth double word, places the value of WSP in WFP and AC3. Increments WSP by twice the frame size (a 16-bit, unsigned integer) to reserve the space for local variables. Sets *OVK* to 1, which enables integer overflow. Sets *OVR* to 0.

The structure of the wide return block pushed is as follows:

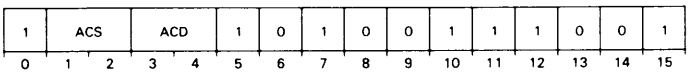
Double Word in Block	Contents
1	PSR, and zeroes in the unused bits
2	AC0
3	AC1
4	AC2
5	Previous WFP
6	Carry and AC3 1–31 (or return PC)

NOTE: *This instruction saves the information required by the* **WRTN** *instruction.*

This instruction is typically executed after an **XJSR** or **LJSR** instruction. Note that neither of these jump instructions can perform a cross ring call. However, they may be used with **WSSVR** to perform an intra-ring transfer to a subroutine that requires no parameters and that uses **WRTN** to return control back to the calling sequence.

Wide Store Byte

WSTB *acs,acd*



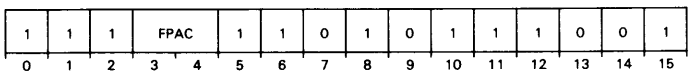
Stores a copy of the rightmost byte of ACD into memory at the address specified by ACS.

ACS contains a 32-bit byte address of some location of memory.

The instruction stores a copy of ACD's bits 24–31 at the locations specified by ACS. The contents of ACS and ACD remain unchanged. Carry is unchanged; *overflow* is 0.

Wide Store Integer

WSTI *fpac*



Converts a floating-point number to an integer and stores it into memory.

AC1 contains the data-type indicator that describes the integer.

AC3 contains a 32-bit byte pointer to a byte in memory. The instruction will store the high-order byte of the number in this location, with the low-order bytes following in subsequent locations.

Under the control of accumulators AC1 and AC3, the instruction translates the contents of the specified FPAC to an integer of the specified type and stores it, right-justified, in memory beginning at the specified location. The instruction leaves the floating-point number unchanged in the FPAC and destroys the previous contents of memory at the specified location(s).

Upon successful completion, the instruction leaves accumulators AC0 and AC1 unchanged. AC2 contains the original contents of AC3. AC3 contains a byte pointer to the first byte following the destination field. Carry is set to 0 and *overflow* is 0.

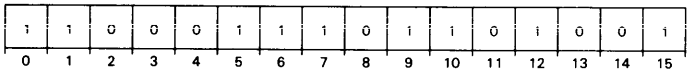
NOTES: *If the number in the specified FPAC has any fractional part, the result of the instruction is undefined. Use the* Integerize *instruction (FINT) to clear any fractional part.*

If the number to be stored is too large to fit in the destination field, this instruction discards high-order digits until the number fits. This instruction stores the remaining low-order digits and sets carry to 1.

If the number to be stored does not completely fill the destination field, the data type of the number determines the instruction's actions. If the number is data type 0, 1, 2, 3, 4, or 5, the instruction sets the high-order bytes to 0. If the number is data type 6, the instruction sign extends it to fill the gap. If the number is data type 7, the instruction sets the low-order bytes to 0.

Wide Store Integer Extended

WSTIX



Converts a floating-point number to an integer and stores it in memory.

AC1 must contain the data-type indicator describing the integer.

AC3 must contain a 32-bit byte pointer pointing to the high-order byte of the destination field in memory.

Using the information in AC1, the instruction converts the contents of each of the FPACs to integer form. Forms a 32-bit integer from the low-order eight digits of each FPAC. Right justifies the integer and stores it in memory beginning at the location specified by AC3. The sign of the integer is the logical OR of the signs of all four FPACs. The previous contents of the addressed memory locations are lost. Sets carry to 0. The contents of the FPACs remain unchanged. The condition codes in the FPSR are unpredictable.

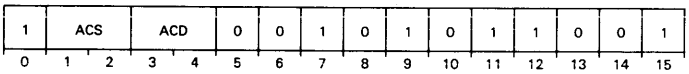
Upon successful termination, the contents of AC0 and AC1 remain unchanged; AC2 contains the original contents of AC3; and AC3 contains a byte pointer pointing to the first byte following the destination field. Carry is set to 0 and *overflow* is 0.

NOTES: *If the integer is too large to fit in the destination field, the instruction discards high-order digits until the integer fits. The instruction stores remaining low-order digits and sets carry to 1.*

If the integer does not completely fill the destination field, the data type of the integer determines the instruction's actions. If the data type is 0, 1, 2, 3, 4, or 5, the instruction sets the high-order bytes to 0. Data types 6 and 7 are illegal and will cause a commercial fault.

Wide Subtract

WSUB *acs,acd*

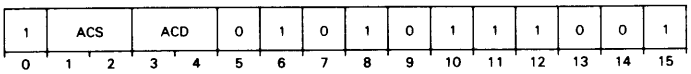


Subtracts two integers contained in accumulators.

Subtracts the 32-bit integer contained in ACS from the 32-bit integer contained in ACD. Stores the result in ACD. Sets carry to the value of ALU carry. Sets *overflow* to 1 if there is an ALU overflow. Unless ACS=ACD, the contents of ACS remain unchanged.

Wide Skip on Zero Bit

WSZB *acs,acd*



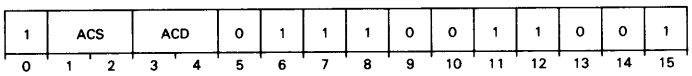
Tests a bit and skips if the bit is 0.

The instruction forms a bit pointer from the contents of ACS and ACD. ACS contains the high-order bits of the bit pointer; ACD contains the low-order bits. ACS and ACD can be specified to be the same accumulator; in this case, the specified accumulator supplies the low-order bits of the bit pointer. The high-order bits are treated as if they were zero in the current ring.

The instruction checks the value of the bit referenced by the bit pointer. If the bit has the value 0, the next sequential word is skipped. If the bit has the value 1, the next sequential word is executed. Carry is unchanged and *overflow* is 0.

Wide Skip on Zero Bit and Set Bit to One

WSZBO *acs,acd*



Tests a bit. Sets the tested bit to 1 and skips if the tested value was 0.

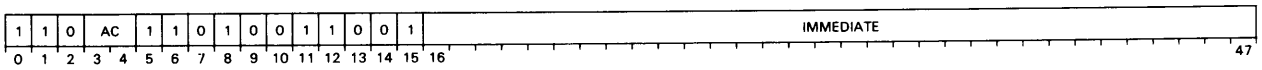
The instruction forms a bit pointer from the contents of ACS and ACD. ACS contains the high-order bits of the bit pointer; ACD contains the low-order bits. ACS and ACD can be specified to be the same accumulator; in this case, the specified accumulator supplies the low-order bits of the bit pointer. The high-order bits are treated as if they were zero.

The instruction checks the value of the bit referenced by the bit pointer. If the bit has the value 0, then the instruction sets the bit to one and skips the next sequential word. If the bit has the value 1, then no skip occurs. Carry is unchanged and *overflow* is 0.

NOTE: *This instruction facilitates the use of bit maps for such purposes as allocation of facilities (memory blocks, I/O devices, etc.) to several processes, or tasks, that may interrupt one another, or in a multiprocessor environment. The bit is tested and set to 1 in one memory cycle.*

Wide Unsigned Skip if AC Greater than Immediate

WUGTI *i,ac*



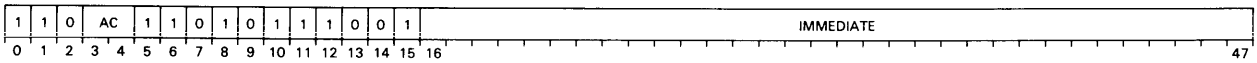
Compares one unsigned integer to another and skips if the first is greater than the second.

Compares the contents of the AC to the 32-bit immediate. If the contents of the AC are greater than the contents of the immediate, then the next sequential word is skipped; otherwise, the next word is executed.

Carry is unchanged and *overflow* is 0.

Wide Unsigned Skip if AC Less than or Equal to Immediate

WULEI *i,ac*



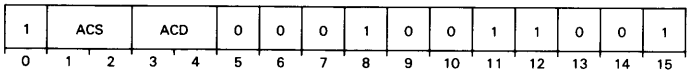
Compares one unsigned integer to another and skips if the first is less than or equal to the second.

Compares the contents of the AC to the 32-bit immediate. If the contents of the AC are less than or equal to the contents of the immediate, then the next sequential word is skipped; otherwise, the next word is executed.

Carry is unchanged and *overflow* is 0.

Wide Unsigned Skip if Greater than or Equal to

WUSGE *acs,acd*



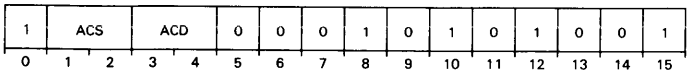
Compares one unsigned integer to another and skips if the first is greater than or equal to the second.

The instruction compares the unsigned, 32-bit integer contained in ACS to the unsigned 32-bit integer in ACD. If the integer contained in ACS is greater than or equal to the integer contained in ACD, the next sequential word is skipped; otherwise, the next sequential word is executed.

If ACS and ACD are the same accumulator, then the instruction will always skip. Carry is unchanged and *overflow* is 0.

Wide Unsigned Skip if Greater than

WUSGT *acs,acd*



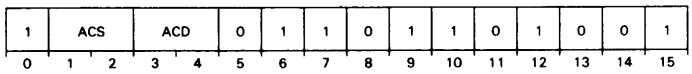
Compares one unsigned integer to another and skips if the first is greater than the second.

The instruction compares the unsigned, 32-bit integer contained in ACS to the unsigned 32-bit integer in ACD. If the integer contained in ACS is greater than the integer contained in ACD, the next sequential word is skipped; otherwise, the next sequential word is executed.

If ACS and ACD are the same accumulator, then the instruction compares the integer contained in the accumulator to 0. The skip will occur if the integer is greater than 0. Carry is unchanged and *overflow* is 0.

Wide Exchange

WXCH *acs,acd*

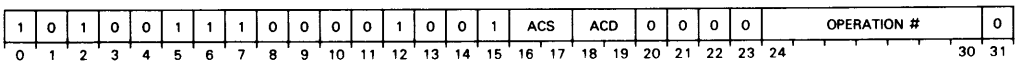


Exchanges the 32-bit contents of ACS and ACD.

Carry is unchanged and *overflow* is 0.

Wide Extended Operation

WXOP *acs,acd,operation #*



The extended operation feature (WXOP) provides an efficient method of transferring control to and from procedures. It allows control to transfer to any 1 of 200 (octal) procedure entry points. WXOP pushes a return block onto the wide stack and transfers control to an extended operation procedure.

The instruction pushes a return block onto the wide stack. Places the address in the wide stack of ACS into AC2; places the address in the wide stack of ACD into AC3. The WXOP table pointer in reserved memory must contain the **WXOP** origin address, the starting address of a 200_8 word table of addresses. These addresses are the starting location of the various **WXOP** operations.

The instruction adds the operation number in the **WXOP** instruction to the **WXOP** origin address to produce the address of a double word in the **WXOP** table. Fetches that word and treats it as the intermediate address in the effective address calculation. After the indirection chain, if any, has been followed, the instruction places the effective address in the program counter. The contents of AC0, AC1, and the **WXOP** origin address remain unchanged. All addresses must be in the current segment. Carry is unchanged and *overflow* is 0.

Figure 10.11 shows the format of the return block pushed by the instruction.

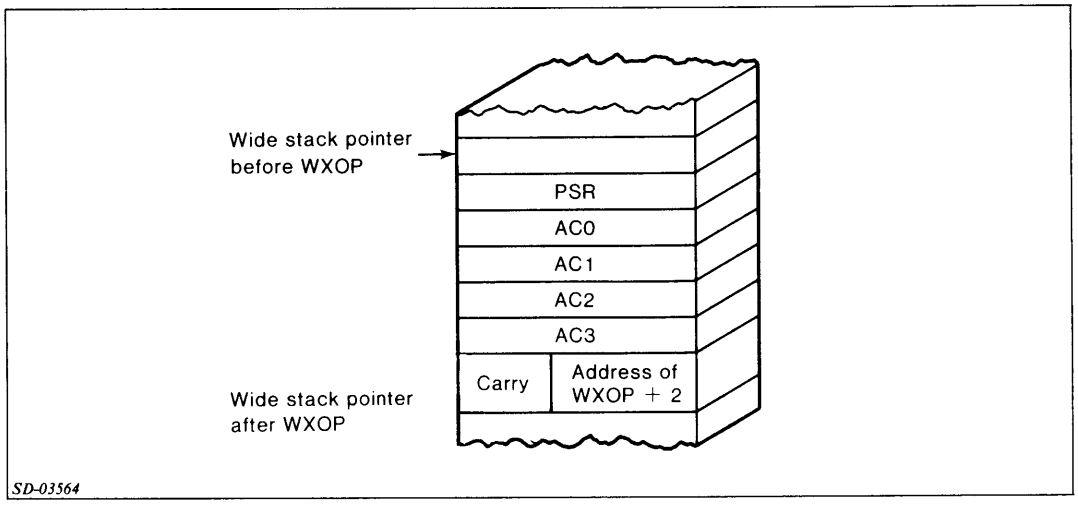
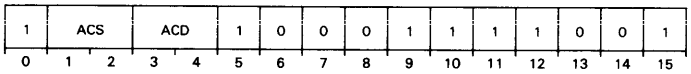


Figure 10.11 WXOP return block pushed

This return block is designed so that the **WXOP** procedure can return control to the calling program via the **WPOPB** instruction.

Wide Exclusive OR

WXOR *acs,acd*

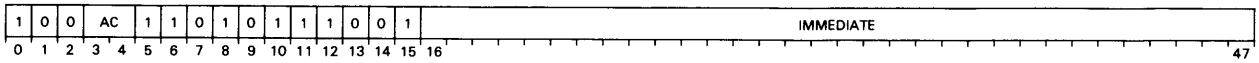


Exclusively ORs the contents of two accumulators.

Forms the logical exclusive OR between corresponding bits of ACS and ACD. Loads the 32-bit result into ACD. The contents of ACS remain unchanged, unless ACS equals ACD. Carry is unchanged and *overflow* is 0.

Wide Exclusive OR Immediate

WXORI *i,ac*

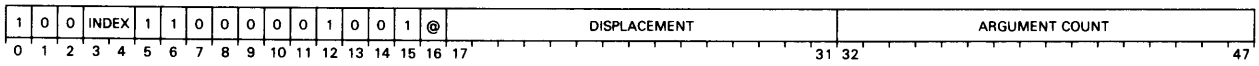


Forms a logical exclusive OR between two values.

The instruction forms the logical exclusive OR between corresponding bits of the specified accumulator and the value contained in the literal field. The instruction places the result of the exclusive OR in the specified accumulator. Carry is unchanged and *overflow* is 0.

Call Subroutine (Extended Displacement)

XCALL *[@]displacement[,index[,argument count]]*



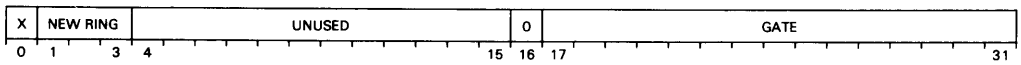
Transfers program control to a subroutine in the current segment or through a gate array in a lower-numbered segment to a subroutine in that lower-numbered segment.

The instruction loads the contents of the PC, plus three, into AC3. The contents of AC3 always reference the current ring.

The effective address (target address) may specify the current ring, an inner ring, or an outer ring.

If the target address specifies an outward ring crossing, a protection fault (code = 7 in AC1) occurs. Note that the contents of the PC in the return block are undefined.

If the target address specifies an inward ring call, then the instruction assumes the target address has the following format:



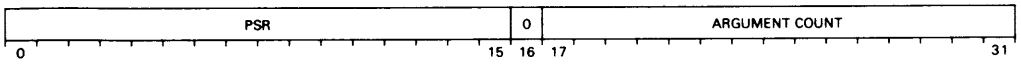
NOTE: *If bit 16 of the inward ring call's target address is not 0 then the results are undefined.*

The instruction checks the gate field of the above format for a legal gate. If the specified gate is illegal, a protection fault (code = 6 in AC1) occurs and no subroutine call is made. Note that the value of the PC in the return block is undefined.

If the specified gate is legal, or if the target address is within the current ring, the instruction then checks the argument count field.

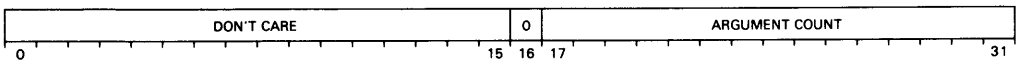
The argument count field specifies the number of arguments the caller has pushed onto the stack. If this field is negative, the caller has also pushed the argument count onto the stack.

If bit 0 of the argument count is 0, the instruction creates a double word with the following format:



The instruction pushes this double word (PSR/argument count) onto the wide stack. If a stack overflow occurs after this push, a stack fault occurs, the PSR is cleared, and no subroutine call is made. Note that the processor uses the inner ring's stack fault handler and the value of the PC in the return block is undefined.

If bit 0 of the argument count is 1 (negative), then the instruction assumes the top double word of the wide stack has the following format:



The instruction uses the argument count on the stack and ignores the argument count coded with the XCALL instruction. The instruction then modifies this double word (PSR/argument count) to include the correct settings of the PSR.

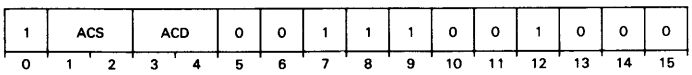
NOTE: *If the target address is in an inner segment, the XCALL instruction copies the number of double words specified in the argument count from the outer segment stack to the inner segment stack, and then pushes the PSR/argument count double word onto the inner stack.*

Regardless of the setting of bit 0 of the argument count, the instruction next unconditionally sets OVR to 0 and loads the PC with the target address. Control then transfers to the word referenced by the PC.

For more information on the XCALL instruction, refer to "Transferring Program Control to Another Segment" in the Program Flow Management chapter.

Exchange Accumulators

XCH *acs,acd*



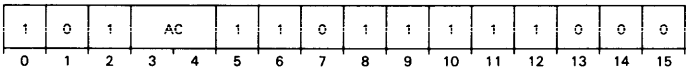
Exchanges the contents of two accumulators.

Places the original contents of bits 16–31 of ACS into bits 16–31 of ACD and the original contents of bits 16–31 of ACD in bits 16–31 of ACS. Carry remains unchanged and *overflow* is 0.

Bits 0–15 of both accumulators are undefined after completion of this instruction.

Execute

XCT *ac*



Executes the contents of an accumulator as an instruction.

Executes the instruction contained in bits 16–31 of the specified accumulator as if it were in main memory in the location occupied by the *Execute* instruction. If the instruction in bits 16–31 of the specified accumulator is an *Execute* instruction that specifies the same accumulator, the processor is placed in a 1-instruction loop.

This instruction leaves carry unchanged; *overflow* is 0.

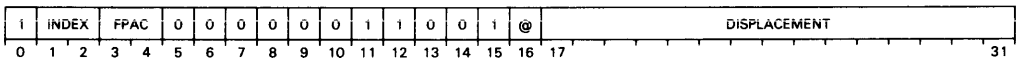
Because of the possibility of bits 16–31 of the specified accumulator containing an *Execute* instruction, this instruction is interruptible. An I/O interrupt can occur immediately prior to each time the instruction in accumulator is executed. If an I/O interrupt does occur, the program counter in the return block pushed on the system stack points to the *Execute* instruction in main memory. This capability to execute an *Execute* instruction gives you a *wait for I/O interrupt* instruction.

> **NOTES:** *If bits 16–31 of the specified accumulator contains the first word of a 2-word instruction, the word following the* **XCT** *instruction is used as the second word. Normal sequential operation then continues from the second word after the* **XCT** *instruction.*
>
> *Do not use the* **XCT** *instruction to execute an instruction that requires all four accumulators, such as* **CMV, CMT, CMP, CTR,** *or* **BAM**.

The results of **XCT** are undefined if bits 16–31 of the specified accumulator contains an instruction that modifies that same accumulator.

Add Double (Memory to FPAC) (Extended Displacement)

XFAMD *fpac,[@]displacement[,index]*

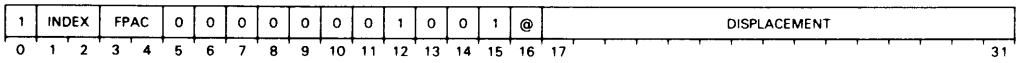


Adds the 64-bit floating-point number in the source location to the 64-bit floating-point number in FPAC and places the normalized result in FPAC.

Computes the effective address, *E*. Uses *E* to address a double-precision (four-word) operand. Adds this 64-bit floating-point number to the floating-point number in the specified FPAC. Places the normalized result in the specified FPAC. Leaves the contents of the source location unchanged and updates the *Z* and *N* flags in the floating-point status register to reflect the new contents of FPAC.

Add Single (Memory to FPAC) (Extended Displacement)

XFAMS *fpac,[@]displacement[,index]*



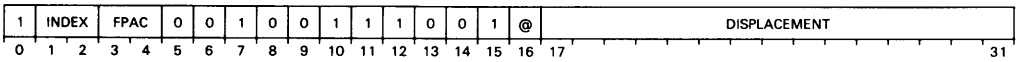
Adds the 32-bit floating-point number in the source location to the 32-bit floating-point number in FPAC and places the normalized result in FPAC.

Computes the effective address, *E*. Uses *E* to address a single-precision (double-word) operand. Adds this 32-bit floating-point number to the floating-point number in bits 0–31 of the specified FPAC. Places the normalized result in the specified FPAC. Leaves the contents of the source location unchanged and updates the *Z* and *N* flags in the floating-point status register to reflect the new contents of FPAC.

Sets bits 32–63 of FACD to 0.

Divide Double (FPAC by Memory) (Extended Displacement)

XFDMD *fpac,[@]displacement[,index]*

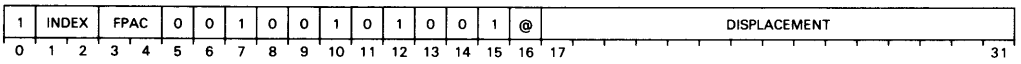


Divides the 64-bit floating-point number in FPAC by the 64-bit floating-point number in the source location and places the normalized result in FPAC.

Computes the effective address, *E*. Uses *E* to address a double-precision (four-word) operand. Divides the floating-point number in the specified FPAC by this 64-bit floating-point number. Places the normalized result in the specified FPAC. Leaves the contents of the source location unchanged and updates the *Z* and *N* flags in the floating-point status register to reflect the new contents of FPAC.

Divide Single (FPAC by Memory) (Extended Displacement)

XFDMS *fpac,[@]displacement[,index]*



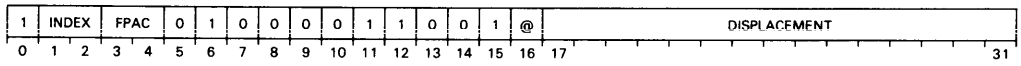
Divides the 32-bit floating-point number in bits 0–31 of FPAC by the 32-bit floating-point number in the source location and places the normalized result in FPAC.

Computes the effective address, *E*. Uses *E* to address a single-precision (double-word) operand. Divides the floating-point number in bits 0–31 of the specified FPAC by this 32-bit floating-point number. Places the normalized result in the specified FPAC. Leaves the contents of the source location unchanged and updates the *Z* and *N* flags in the floating-point status register to reflect the new contents of FPAC.

Sets bits 32–63 of FACD to 0.

Load Floating-Point Double (Extended Displacement)

XFLDD *fpac,[@]displacement[,index]*



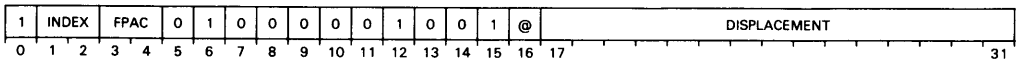
Moves four words out of memory and into a specified FPAC.

Computes the effective address, *E*. Fetches the double-precision floating-point number at the address specified by *E* and places it in FPAC. Updates the *Z* and *N* flags in the FPSR to reflect the new contents of FPAC.

NOTE: *This instruction will move unnormalized data without change, but the* Z *and* N *flags will be undefined.*

Load Floating-Point Single (Extended Displacement)

XFLDS *fpac,[@]displacement[,index]*



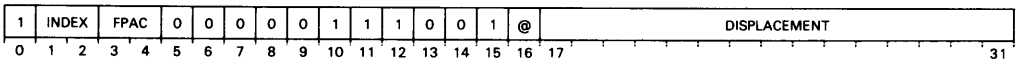
Moves two words out of memory into a specified FPAC.

Computes the effective address, *E*. Fetches the single-precision floating-point number at the address specified by *E*. Places the number in the high-order bits of FPAC. Sets the low-order 32 bits of FPAC to 0. Updates the *Z* and *N* flags in the floating-point status register to reflect the new contents of FPAC.

NOTE: *This instruction will move unnormalized data without change, but the* Z *and* N *flags will be undefined.*

Multiply Double (FPAC by Memory) (Extended Displacement)

XFMMD *fpac,[@]displacement[,index]*

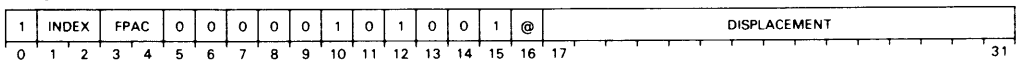


Multiplies the 64-bit floating-point number in the source location by the 64-bit floating-point number in FPAC and places the normalized result in FPAC.

Computes the effective address, *E*. Uses *E* to address a double-precision (four-word) operand. Multiplies this 64-bit floating-point number by the floating-point number in the specified FPAC. Places the normalized result in the specified FPAC. Leaves the contents of the source location unchanged and updates the *Z* and *N* flags in the floating-point status register to reflect the new contents of FPAC.

Multiply Single (FPAC by Memory) (Extended Displacement)

XFMMS *fpac,[@]displacement[,index]*

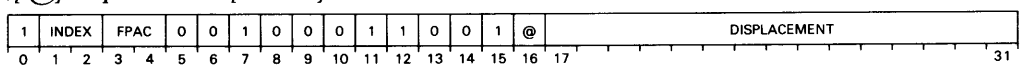


Multiplies the 32-bit floating-point number in the source location by the 32-bit floating-point number in bits 0–31 of FPAC and places the normalized result in FPAC.

Computes the effective address, *E*. Uses *E* to address a single-precision (double-word) operand. Multiplies this 32-bit floating-point number by the floating-point number in bits 0–31 of the specified FPAC. Places the normalized result in bits 0–31 of the specified FPAC. Sets bits 32–63 of FPAC to 0. Leaves the contents of the source location unchanged and updates the *Z* and *N* flags in the floating-point status register to reflect the new contents of FPAC.

Subtract Double (Memory from FPAC) (Extended Displacement)

XFSMD *fpac,[@]displacement[,index]*

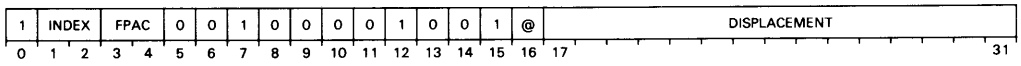


Subtracts the 64-bit floating-point number in the source location from the 64-bit floating-point number in FPAC and places the normalized result in FPAC.

Computes the effective address, *E*. Uses *E* to address a double-precision (four-word) operand. Subtracts this 64-bit floating-point number from the floating-point number in the specified FPAC. Places the normalized result in the specified FPAC. Leaves the contents of the source location unchanged and updates the *Z* and *N* flags in the floating-point status register to reflect the new contents of FPAC.

Subtract Single (Memory from FPAC) (Extended Displacement)

XFSMS *fpac,[@]displacement[,index]*

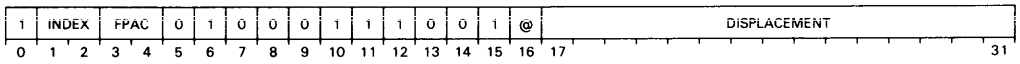


Subtracts the 32-bit floating-point number in the source location from the 32-bit floating-point number in bits 0–31 of FPAC and places the normalized result in FPAC.

Computes the effective address, *E*. Uses *E* to address a single-precision (double-word) operand. Subtracts this 32-bit floating-point number from the floating-point number in bits 0–31 of the specified FPAC. Places the normalized result in the specified FPAC. Sets bits 32–63 of FPAC to 0. Leaves the contents of the source location unchanged and updates the *Z* and *N* flags in the floating-point status register to reflect the new contents of FPAC.

Store Floating-Point Double (Extended Displacement)

XFSTD *fpac,[@]displacement[,index]*



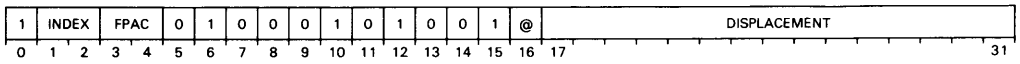
Stores the contents of a specified FPAC into a memory location.

Computes the effective address, *E*. Places the floating-point number contained in FPAC in memory beginning at the location addressed by *E*. Destroys the previous contents of the addressed memory location. The contents of FPAC and the condition codes in the FPSR remain unchanged.

NOTE: *This instruction will move unnormalized data without change.*

Store Floating-Point Single (Extended Displacement)

XFSTS *fpac,[@]displacement[,index]*



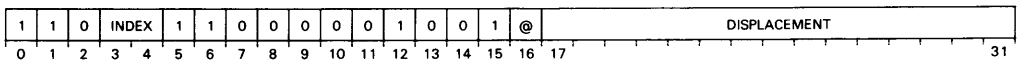
Stores the contents of a specified FPAC into a memory location.

Computes the effective address, *E*. Places the 32 high-order bits of FPAC in memory beginning at the location addressed by *E*. Destroys the previous contents of the addressed memory location. The contents of FPAC and the condition codes in the FPSR remain unchanged.

NOTE: *This instruction will move unnormalized or data without change.*

Jump (Extended Displacement)

XJMP *[@]displacement[,index]*



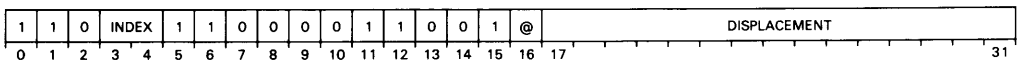
Loads an effective address into the program counter.

Calculates the effective address, *E*. Loads *E* into the PC. Carry is unchanged and *overflow* is 0.

NOTE: *The calculation of* E *is forced to remain within the current segment of execution.*

Jump to Subroutine (Extended Displacement)

XJSR *[@]displacement[,index]*



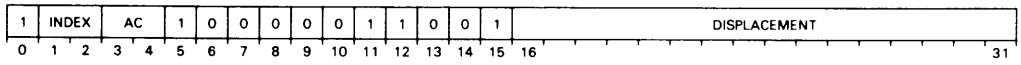
Saves a return address and transfers control to a subroutine.

Calculates the effective address, *E*. Loads the current value of the PC, plus two, into AC3. Loads *E* into the PC. Carry is unchanged and *overflow* is 0.

NOTE: *The calculation of* E *is forced to remain within the current segment of execution.*

Load Byte (Extended Displacement)

XLDB *ac,displacement[,index]*

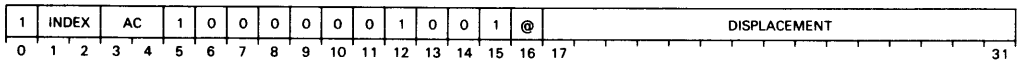


Calculates an effective byte address and loads the byte into the specified accumulator.

Calculates the effective byte address. Uses the byte address to reference a byte in memory. Loads the addressed byte into bits 24–31 of the specified accumulator, then zero extends the value to 32 bits. Carry is unchanged and *overflow* is 0.

Load Effective Address (Extended Displacement)

XLEF *ac,[@]displacement[,index]*



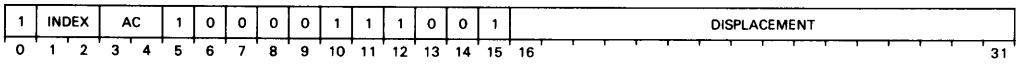
Loads an effective address into an accumulator.

Calculates the effective address, *E*. Loads *E* into the specified accumulator. Carry is unchanged and *overflow* is 0.

NOTE: *Bit 0 of the result is guaranteed to be 0.*

Load Effective Byte Address (Extended Displacement)

XLEFB *ac,displacement[,index]*



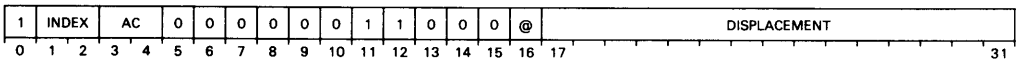
Loads an effective byte address into an accumulator.

Calculates the effective byte address. Loads the byte address into the specified accumulator. Carry is unchanged and *overflow* is 0.

NOTE: *Bit 0 of the result is guaranteed to be 0.*

Narrow Add Memory Word to Accumulator (Extended Displacement)

XNADD *ac,[@]displacement[,index]*

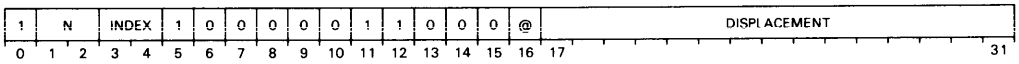


Adds an integer contained in a memory location to an integer contained n an accumulator.

Calculates the effective address, *E*. Adds the 16-bit integer contained in the location specified by *E* to the integer contained in bits 16–31 of the specified accumulator. Sign extends the 16-bit result to 32 bits and loads it into the specified accumulator. Sets carry to the value of ALU carry and *overflow* to 1, if there is an ALU overflow. The contents of the referenced memory location remain unchanged.

Narrow Add Immediate (Extended Displacement)

XNADI *n,[@]displacement[,index]*



Adds an integer in the range of 1 to 4 to an integer contained in a 16-bit memory location.

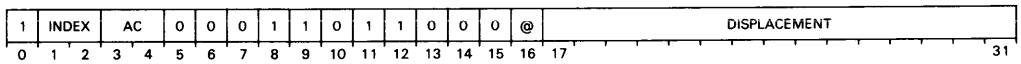
Adds the value $n+1$ to the 16-bit contents of the specified memory location, where n is an integer in the range of 0 to 3. Sets carry to the value of ALU carry (16-bit operation). Sets *overflow* to 1, if there is an ALU overflow (16-bit operation).

NOTE: *The assembler takes the coded value of* n *and subtracts one from it before placing it in the immediate field. Therefore, the programmer should code the exact value to be added.*

The operation performed by this instruction is not indivisible.

Narrow Divide Memory Word (Extended Displacement)

XNDIV *ac,[@]displacement[,index]*

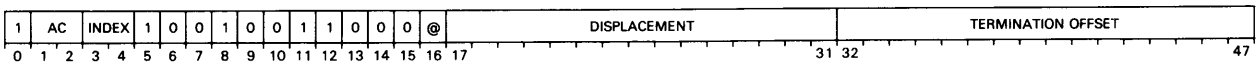


Divides an integer contained in an accumulator by an integer in memory.

Calculates the effective address, *E*. Sign extends the integer contained in bits 16–31 of the specified accumulator to 32 bits and divides it by the 16-bit integer contained in the location specified by *E*. If the quotient is within the range −32,768 to +32,767 inclusive, sign extends the result to 32 bits and loads it into the specified accumulator. If the quotient is outside of this range, or the memory word is zero, the instruction sets *overflow* to 1 and leaves the specified accumulator unchanged. Otherwise, *overflow* is 0. The contents of the referenced memory location and carry remain unchanged.

Narrow Do Until Greater than (Extended Displacement)

XNDO *ac,termination offset,[@]displacement[,index]*



Increments a memory location, compares it to the AC, and takes a normal exit if the location is still less than or equal to the AC.

Increments a 16-bit memory location, sign extends it to 32 bits, and compares it to the AC.

If the memory location is greater than the AC, then a PC relative branch is made by adding the termination offset to PC+1.

If the memory location is less than or equal to the AC, then the next instruction is executed.

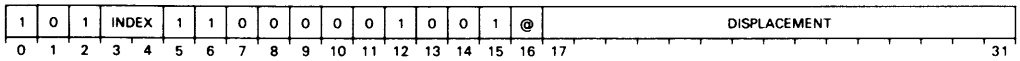
In either case, the instruction loads the incremented memory location into the AC.

If a fixed-point overflow trap occurs while incrementing the DO-loop variable, the contents of the specified memory location and the PC value in the return block are undefined.

Sets carry to the value of ALU carry. Sets *overflow* to 1, if there is an ALU overflow caused by the increment.

Narrow Decrement and Skip if Zero (Extended Displacement)

XNDSZ *[@]displacement[,index]*



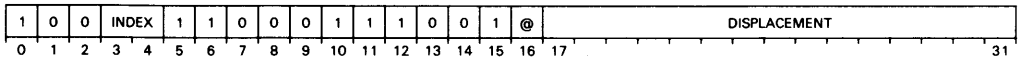
Decrements the contents of a location and skips the next word if the decremented value is zero.

Calculates the effective address, *E*. Decrements by one the contents of the 16-bit memory location addressed by *E*. If the result is equal to zero, then the instruction skips the next sequential word. Carry is unchanged and *overflow* is 0.

NOTE: *This instruction is indivisible.*

Narrow Increment and Skip if Zero (Extended Displacement)

XNISZ *[@]displacement[,index]*



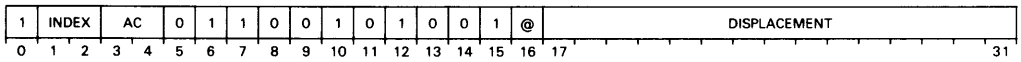
Increments the contents of a location and skips the next word if the incremented value is zero.

Calculates the effective address, *E*. Increments by one the contents of the 16-bit memory location addressed by *E*. If the result is equal to zero, then the instruction skips the next sequential word. Carry is unchanged and *overflow* is 0.

NOTE: *This instruction is indivisible.*

Narrow Load Accumulator (Extended Displacement)

XNLDA *ac,[@]displacement[,index]*

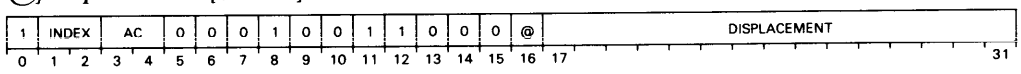


Sign extends and loads the contents of a memory location into an accumulator.

Calculates the effective address, *E*. Fetches the 16-bit fixed-point integer contained in the location specified by *E*. Sign extends this integer to 32 bits and loads it into the specified accumulator. Carry is unchanged and *overflow* is 0.

Narrow Multiply Memory Word (Extended Displacement)

XNMUL *ac,[@]displacement[,index]*

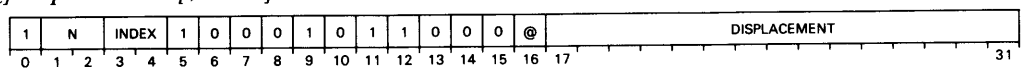


Multiplies an integer in memory by an integer in an accumulator.

Calculates the effective address, *E*. Multiplies the 16-bit, signed integer contained in the location referenced by *E* by the signed integer contained in bits 16–31 of the specified accumulator. If the result is outside the range of −32,768 to +32,767 inclusive, sets *overflow* to 1; otherwise, *overflow* is 0. Sign extends the result to 32 bits and places the result in the specified accumulator. The contents of the referenced memory location and carry remain unchanged.

Narrow Subtract Immediate (Extended Displacement)

XNSBI *n,[@]displacement[,index]*



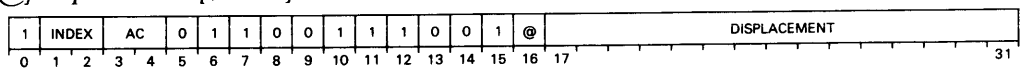
Subtracts an integer in the range of 1 to 4 from an integer contained in a 16-bit memory location.

Subtracts the value $n+1$ from the 16-bit value contained in the specified memory location, where *n* is an integer in the range of 0 to 3. Sets carry to the value of ALU carry (16-bit operation). Sets *overflow* to 1, if there is an ALU overflow (16-bit operation).

NOTE: *The assembler takes the coded value of* n *and subtracts one from it before placing it in the immediate field. Therefore, the programmer should code the exact value to be subtracted.*

Narrow Store Accumulator (Extended Displacement)

XNSTA *ac,[@]displacement[,index]*

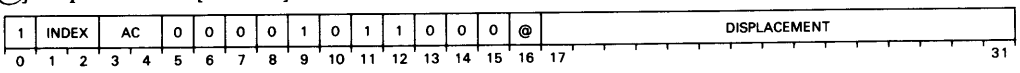


Stores the contents of an accumulator in a memory location.

Calculates the effective address, *E*. Stores a copy of the 16-bit contents of the specified accumulator in the memory location specified by *E*. Carry is unchanged; *overflow* is 0.

Narrow Subtract Memory Word (Extended Displacement)

XNSUB *ac,[@]displacement[,index]*

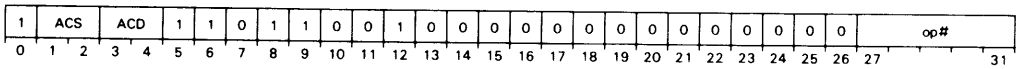


Subtracts an integer in memory from an integer in an accumulator.

Calculates the effective address, *E*. Subtracts the 16-bit integer contained in the location referenced by *E* from the integer contained in bits 16–31 of the specified accumulator. Sign extends the result to 32 bits and stores it in the specified accumulator. Sets carry to the value of ALU carry and *overflow* to 1, if there is an ALU overflow. The contents of the specified memory location remain unchanged.

Extended Operation

XOP0 *acs,acd,operation #*



Pushes a return block onto the narrow stack and transfers control to an extended operation procedure.

Pushes a return block onto the narrow stack. Places the address in the narrow stack of ACS into AC2; places the address in the narrow stack of ACD into AC3. The XOP table pointer in reserved memory must contain the **XOP0** origin address, the starting address of a 40_8 word table of addresses. These addresses are the starting location of the various **XOP0** operations.

Adds the operation number in the **XOP0** instruction to the **XOP0** origin address to produce the address of a word in the **XOP0** table. Fetches that word and treats it as the intermediate address in the effective address calculation. After the indirection chain, if any, has been followed, the instruction places the effective address in the program counter. The contents of carry, AC0, AC1, and the **XOP0** origin address remain unchanged. *Overflow* is 0.

Figure 10.12 shows the format of the return block pushed by the instruction.

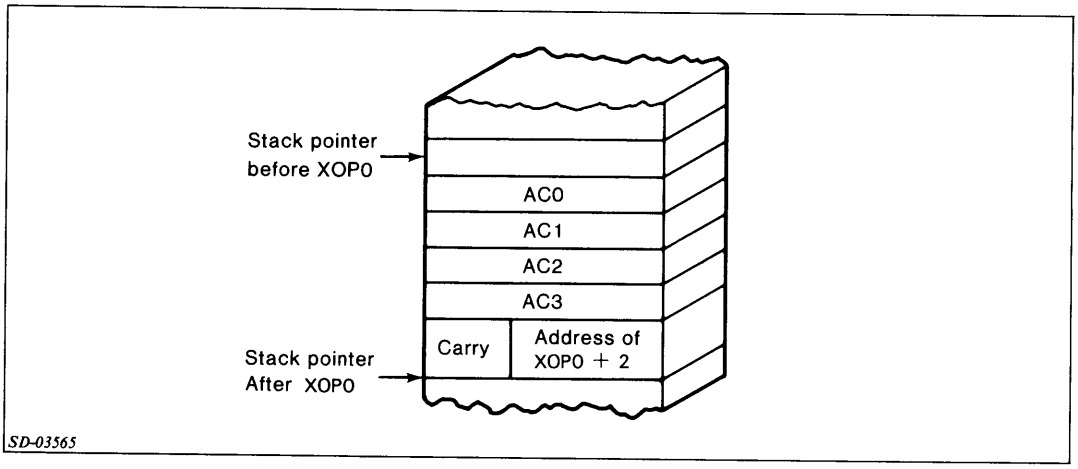
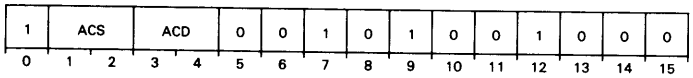


Figure 10.12 XOP0 return block

This return block is designed so that the **XOP0** procedure can return control to the calling program via the *Pop Block* instruction.

Exclusive OR

XOR *acs,acd*



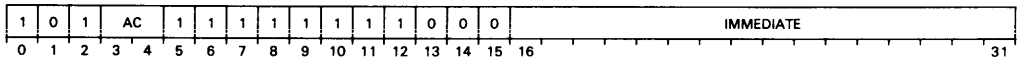
Exclusively ORs the contents of two accumulators.

Forms the logical exclusive OR of the contents of bits 16–31 of ACS and the contents of bits 16–31 of ACD and places the result in bits 16–31 of ACD. Sets a bit position in the result to 1 if the corresponding bit positions in the two operands are unlike; otherwise, the instruction sets result bit to 0. The contents of ACS and carry remain unchanged. *Overflow* is 0.

Bits 0–15 of the modified accumulator are undefined after completion of this instruction.

Exclusive OR Immediate

XORI *i,ac*



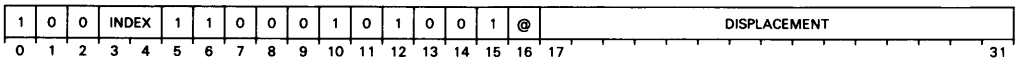
Exclusively ORs the contents of an accumulator with the contents of a 16-bit number in the instruction.

Forms the logical exclusive OR of the contents of the immediate field and the contents of bits 16–31 of the specified accumulator and places the result in bits 16–31 of the specified accumulator. Carry remains unchanged and *overflow* is 0.

Bits 0–15 of the modified accumulator are undefined after completion of this instruction.

Push Address (Extended Displacement)

XPEF *[@]displacement[,index]*



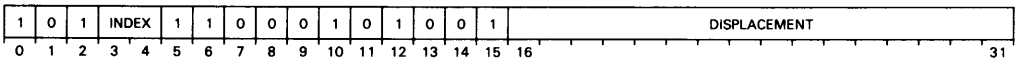
Pushes an address onto the wide stack.

Calculates the effective address, *E*. Pushes *E* onto the wide stack, then checks for stack overflow. Carry is unchanged and *overflow* is 0.

NOTE: *Bit 0 of the pushed result is guaranteed to be 0.*

Push Byte Address (Extended Displacement)

XPEFB *displacement[,index]*

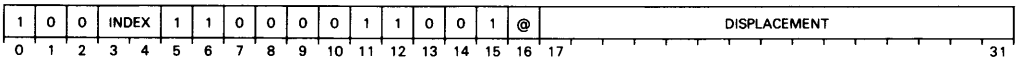


Pushes an effective byte address onto the wide stack.

Calculates a 32-bit byte address. Pushes this byte address onto the wide stack, then checks for stack overflow. Carry is unchanged and *overflow* is 0.

Push Jump (Extended Displacement)

XPSHJ *[@]displacement[,index]*



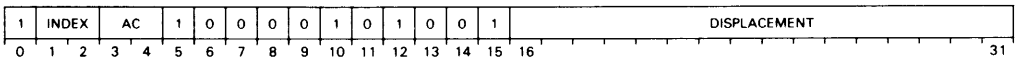
Pushes the program counter onto the wide stack and jumps to a subroutine.

Calculates the effective address, *E*. Pushes the current 31-bit value of the PC plus two onto the wide stack. Loads the PC with *E*. Checks for stack overflow. Carry is unchanged and *overflow* is 0.

NOTE: *The address pushed onto the wide stack will always reference the current segment.*

Store Byte (Extended Displacement)

XSTB *ac,displacement[,index]*

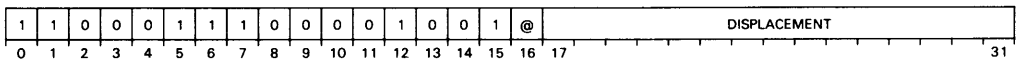


Stores the low-order byte of the specified accumulator in memory.

Calculates the effective byte address. Moves a copy of the contents of bits 24–31 of the specified accumulator into memory at the location specified by the byte address. Carry is unchanged and *overflow* is 0.

Vector on Interrupting Device (Extended Displacement)

XVCT *[@]displacement*



The XVCT instruction must execute in segment zero. A protection violation occurs if XVCT executes in another segment.

When a device requests an interrupt, the processor fetches the first instruction that the interrupt handler address references. For a type 3 interrupt handler, the XVCT instruction must be the first instruction the processor fetches. The processor executes it before honoring further interrupts.

The effective address, produced by the evaluation of the absolute displacement, refers to entry zero of the vector table in segment zero. (The indirection chain, if any, is narrow.) The interrupting device number becomes a double-word offset that points to the appropriate entry. Bits 1-31 of the vector table entry contain the address of entry zero of the device control table for the interrupting device.

The processor saves the wide stack register contents and the wide stack fault handler address, and initializes the wide stack registers and fault handler for the vector stack and the vector stack fault handler. The processor pushes the old wide registers, the old fault handler address, a wide return block, and the old mask onto the new stack.

The processor initializes AC0, AC1, AC2, PSR, and PC.

AC0 contains the revised priority mask. The processor uses this mask to perform a maskout.

AC1 contains the I/O channel and the device code in bits 23-31; zero extended.

AC2 contains the entry zero address of the device control table.

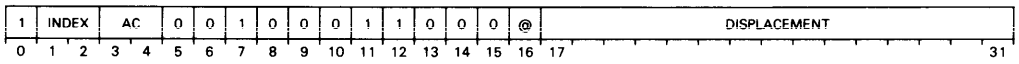
PSR contains the PSR word from word four of the device control table.

PC contains the address of the device interrupt routine from bits 4–31 of the first double word (words zero and one) of the device control table. The processor transfers control to the word addressed by the program counter.

Refer to Interrupt Servicing in the Device Management chapter for further information.

Wide Add Memory Word to Accumulator (Extended Displacement)

XWADD *ac,[@]displacement[,index]*

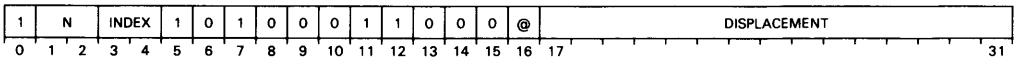


Adds an integer contained in memory to an integer contained in an accumulator.

Calculates the effective address, *E*. Adds the 32-bit integer contained in the location specified by *E* to the 32-bit integer contained in the specified accumulator. Loads the result into the specified accumulator. Sets carry to the value of ALU carry and *overflow* to 1, if there is an ALU overflow. The contents of the referenced memory location remain unchanged.

Wide Add Immediate (Extended Displacement)

XWADI *n,[@]displacement[,index]*



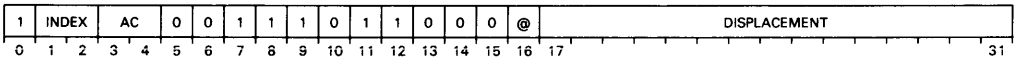
Adds an integer in the range of 1 to 4 to an integer contained in a 32-bit memory location.

Adds the value $n+1$ to the 32-bit fixed-point integer contained in a memory location, where *n* is an integer in the range of 0 to 3. Sets carry to the value of ALU carry. Sets *overflow* to 1, if there is an ALU overflow.

> **NOTE:** *The assembler takes the coded value of* n *and subtracts one from it before placing it in the immediate field. Therefore, the programmer should code the exact value to be added.*

Wide Divide Memory Word (Extended Displacement)

XWDIV *ac,[@]displacement[,index]*



Divides an integer in an accumulator by an integer in memory.

Calculates the effective address, *E*. Sign extends the 32-bit integer contained in the specified accumulator to 64 bits and divides it by the 32-bit integer contained in the location specified by *E*.

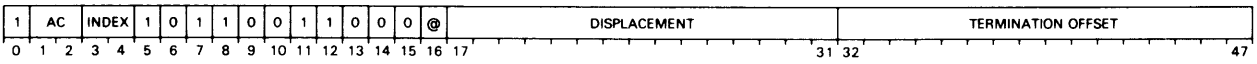
If the quotient is within the range of $-2{,}147{,}483{,}648$ to $+2{,}147{,}483{,}647$ inclusive, the instruction loads the quotient into the specified accumulator. *Overflow* is 0.

If the quotient is outside this range, or if the word in memory is zero, the instruction sets *overflow* to 1 and leaves the specified accumulator unchanged.

The contents of the referenced memory location and carry remain unchanged.

Wide Do Until Greater than (Extended Displacement)

XWDO *ac,termination offset,[@]displacement[,index]*



Increments a memory location, compares it to the AC, and takes a normal exit if the location is still less than or equal to the AC.

Increments a 32-bit memory location and compares it to the AC.

If the memory location is greater than the AC, then a PC relative branch is made by adding the termination offset to PC+1.

If the memory location is less than or equal to the AC, then the next instruction is executed.

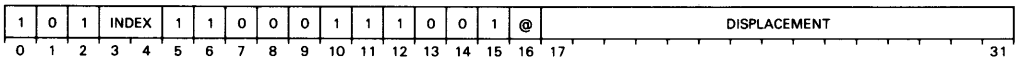
In either case, the instruction loads the incremented memory location into the AC.

If a fixed-point overflow trap occurs while incrementing the DO-loop variable, the contents of the specified memory location and the PC value in the return block are undefined.

Sets carry to the value of ALU carry. Sets *overflow* to 1, if there is an ALU overflow caused by the increment.

Wide Decrement and Skip if Zero (Extended Displacement)

XWDSZ *[@]displacement[,index]*



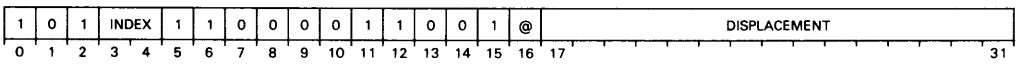
Decrements the contents of a location and skips the next word if the decremented value is zero.

Calculates the effective address, *E*. Decrements by one the contents of the 32-bit memory location addressed by *E*. If the result is equal to zero, then the instruction skips the next sequential word. Carry is unchanged and *overflow* is 0.

NOTE: *This instruction executes in one indivisible memory cycle if the word to be decremented is located on a double-word boundary.*

Wide Increment and Skip if Zero (Extended Displacement)

XWISZ *[@]displacement[,index]*



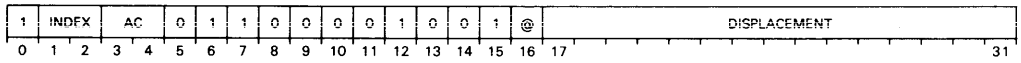
Increments the contents of a location and skips the next word if the incremented value is zero.

Calculates the effective address, *E*. Increments by one the contents of the 32-bit memory location addressed by *E*. If the result is equal to zero, then the instruction skips the next sequential word. Carry is unchanged and *overflow* is 0.

NOTE: *This instruction executes in one indivisible memory cycle if the word to be incremented is located on a double-word boundary.*

Wide Load Accumulator (Extended Displacement)

XWLDA *ac,[@]displacement[,index]*

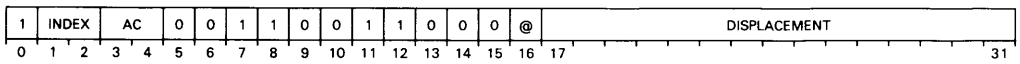


Loads the contents of a memory location into an accumulator.

Calculates the effective address, *E*. Fetches the 32-bit fixed-point integer contained in the location specified by *E*. Loads a copy of this integer into the specified accumulator. Carry is unchanged and *overflow* is 0.

Wide Multiply Memory Word (Extended Displacement)

XWMUL *ac,[@]displacement[,index]*



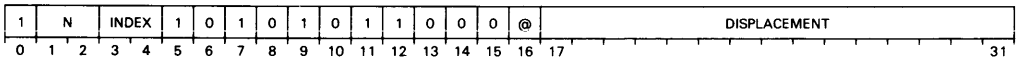
Multiplies an integer in an accumulator by an integer in memory.

Calculates the effective address, *E*. Multiplies the 32-bit, signed integer contained in the location referenced by *E* by the 32-bit, signed integer contained in the specified accumulator. Loads the 32 least significant bits of the result into the specified accumulator.

If the result is outside the range of −2,147-483,648 to +2,147,483,647 inclusive, the instruction sets *overflow* to 1; otherwise, *overflow* is 0. The contents of the referenced memory location and carry remain unchanged.

Wide Subtract Immediate (Extended Displacement)

XWSBI *n,[@]displacement[,index]*



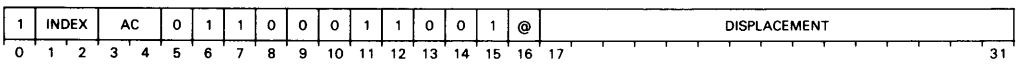
Subtracts an integer in the range of 1 to 4 from an integer contained in a 32-bit memory location.

Subtracts the value $n+1$ from the value contained in the specified 32-bit memory location, where *n* is an integer in the range of 0 to 3. Sets carry to the value of ALU carry. Sets *overflow* to 1, if there is an ALU overflow.

> **NOTE:** *The assembler takes the coded value of* n *and subtracts one from it before placing it in the immediate field. Therefore, the programmer should code the exact value to be subtracted.*

Wide Store Accumulator (Extended Displacement)

XWSTA *ac,[@]displacement[,index]*

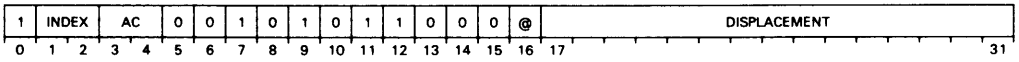


Stores the contents of an accumulator in a memory location.

Calculates the effective address, *E*. Stores a copy of the 32-bit contents of the specified accumulator in the memory location specified by *E*. Carry is unchanged; *overflow* is 0.

Wide Subtract Memory Word (Extended Displacement)

XWSUB *ac,[@]displacement[,index]*

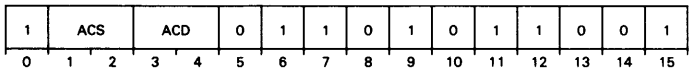


Subtracts an integer contained in memory from an integer contained in an accumulator.

Calculates the effective address, *E*. Subtracts the 32-bit integer contained in the location referenced by *E* from the 32-bit integer contained in the specified accumulator. Loads the result into the specified accumulator. Sets carry to the value of ALU carry and *overflow* to 1, if there is an ALU overflow. The contents of the specified memory location remain unchanged.

Zero Extend

ZEX *acs,acd*



Zero extends a 16-bit integer in an accumulator to 32 bits.

Zero extends the 16-bit integer contained in ACS to 32 bits and loads the result into ACD. The contents of ACS remain unchanged, unless ACS equals ACD. Carry is unchanged and *overflow* is 0.

Appendix A
Anomalies

C/350 Instruction Results

Bits 0-15 of results produced by C/350 instructions are undefined unless stated otherwise. When these instructions are used in a 32-bit environment, you should zero extend (ZEX) or sign extend (SEX) the results as appropriate. Notable exceptions to this rule are the LEF and FRH instructions.

Wide Instruction Opcodes

The processor recognizes the wide instructions supported by this machine by the instruction opcodes. The instructions are an outgrowth of the C/350 ALC no load-always skip opcode and the C/350 XOP and XOP1 opcodes. This means that on this machine you cannot use any C/350 program that contains these instructions. The processor will interpret these instructions as wide instructions, not as C/350 instructions.

Program Counter Wraparound

The program counter is 31 bits wide. Bits 1-3 specify the current segment of execution. Bits 4-31 specify an address. When the program counter is incremented, only bits 4-31 take part in the increment. This means that the program counter will always contain an address in the current segment. Program counter wraparound will *not* occur at 77777_8 as it does in the C/350.

Float/Fixed Conversions

When the processor converts a floating-point number to a fixed-point integer, it converts the largest negative number correctly without MOF overflow. For single precision, the processor converts the integer portion of floating-point numbers to an integer in the range -32,768 to +32,767 inclusive. For double precision, the processor converts the integer portion to an integer in the range of -2,147,483,648 to +2,147,483,647 inclusive.

Address Wraparound

When using the C/350 BAM, BLM, CMP, CMT, CMV, CTR, and EDIT instructions, address wraparound may not occur at 77777_8. This means that a C/350 program counter can possibly generate logical addresses larger than 64 kbytes. In this situation, results are undefined.

If any of the instructions listed in the previous paragraph move data backwards (i.e., in descending addresses) and cross a segment boundary, a protection fault occurs. AC1 will contain the protection code 4.

C/350 Signed Divide Instructions

When the C/350 DIVS or DIVX instructions produce a result of -32,768, the 32-bit processor sets carry to zero (meaning no overflow). When this instruction is used on the C/350, the processor sets CARRY to one (meaning overflow). Wide divide instruction set *overflow* to zero when -32,768 results.

NIO CPU Instructions

The NIO*[f]* CPU instructions are reserved or assigned a function. For instance, the NIOS CPU is the interrupt enable instruction (INTEN).

Floating-Point Trap

The 32-bit processor responds to floating-point traps upon completion of the floating-point instruction that caused the fault. In the C/350, the response to a floating-point trap occurs when the *next* floating-point instruction is encountered. In either case, the value of the floating-point PC (FPPC) is the same; that is, it contains the address of the first floating-point instruction that caused a fault.

Floating-Point Numerical Algorithms

The C/350 floating-point loads (FLDS, FLDD) do not correct impure zero input. All loads simply move the memory operand to the specified FPAC. No normalization and correction to true zero is performed. The *Z* and *N* bits of the FPSR are set to reflect the loaded operand only if the operand is normalized. The *Z* and *N* flags are undefined if the operand is unnormalized.

For all instructions, *true zero* is guaranteed to be generated for valid inputs *only*. If an impure zero is generated with invalid inputs, the result is not necessarily converted to a true zero.

The C/350 FFAS and FFMD instructions leave the *Z* and *N* bits of the FPSR unchanged.

Otherwise, when bit 8 of the FPSR is a zero, the results of the floating-point computation performed on the 32-bit processor are identical to those obtained on the C/350.

C/350 Commercial Faults

A C/350 commercial fault loads different information in AC0, AC2, and AC3 after the fault is taken. The size of the return block, the fault code in AC1, and the meaning of the PC in the return block are identical to the results obtained on the C/350.

Appendix B
ASCII Codes

DECIMAL	OCTAL	HEX	KEY SYMBOL	MNEMONIC
0	000	00	↑@	NUL
1	001	01	↑A	SOH
2	002	02	↑B	STX
3	003	03	↑C	ETX
4	004	04	↑D	EOT
5	005	05	↑E	ENQ
6	006	06	↑F	ACK
7	007	07	↑G	BEL
8	010	08	↑H	BS (BACKSPACE)
9	011	09	↑I	TAB
10	012	0A	↑J	NEW LINE
11	013	0B	↑K	VT (VERT. TAB)
12	014	0C	↑L	FORM FEED
13	015	0D	↑M	CARRIAGE RETURN
14	016	0E	↑N	SO
15	017	0F	↑O	SI
16	020	10	↑P	DLE
17	021	11	↑Q	DC1
18	022	12	↑R	DC2
19	023	13	↑S	DC3
20	024	14	↑T	DC4
21	025	15	↑U	NAK
22	026	16	↑V	SYN
23	027	17	↑W	ETB
24	030	18	↑X	CAN
25	031	19	↑Y	EM
26	032	1A	↑Z	SUB
27	033	1B	ESC	ESCAPE
28	034	1C	↑\	FS
29	035	1D	↑]	GS
30	036	1E	↑↑	RS
31	037	1F	↑←	US

DECIMAL	OCTAL	HEX	KEY SYMBOL
32	040	20	SPACE
33	041	21	!
34	042	22	" (QUOTE)
35	043	23	#
36	044	24	$
37	045	25	%
38	046	26	&
39	047	27	' (APOS)
40	050	28	(
41	051	29	)
42	052	2A	*
43	053	2B	+
44	054	2C	, (COMMA)
45	055	2D	-
46	056	2E	. (PERIOD)
47	057	2F	/
48	060	30	0
49	061	31	1
50	062	32	2
51	063	33	3
52	064	34	4
53	065	35	5
54	066	36	6
55	067	37	7
56	070	38	8
57	071	39	9
58	072	3A	:
59	073	3B	;
60	074	3C	<
61	075	3D	=
62	076	3E	>
63	077	3F	?
64	100	40	@

DECIMAL	OCTAL	HEX	KEY SYMBOL
65	101	41	A
66	102	42	B
67	103	43	C
68	104	44	D
69	105	45	E
70	106	46	F
71	107	47	G
72	110	48	H
73	111	49	I
74	112	4A	J
75	113	4B	K
76	114	4C	L
77	115	4D	M
78	116	4E	N
79	117	4F	O
80	120	50	P
81	121	51	Q
82	122	52	R
83	123	53	S
84	124	54	T
85	125	55	U
86	126	56	V
87	127	57	W
88	130	58	X
89	131	59	Y
90	132	5A	Z
91	133	5B	[
92	134	5C	\
93	135	5D	]
94	136	5E	↑ (OR) ^
95	137	5F	← (OR) _
96	140	60	` (GRAVE)

DECIMAL	OCTAL	HEX	KEY SYMBOL
97	141	61	a
98	142	62	b
99	143	63	c
100	144	64	d
101	145	65	e
102	146	66	f
103	147	67	g
104	150	68	h
105	151	69	i
106	152	6A	j
107	153	6B	k
108	154	6C	l
109	155	6D	m
110	156	6E	n
111	157	6F	o
112	160	70	p
113	161	71	q
114	162	72	r
115	163	73	s
116	164	74	t
117	165	75	u
118	166	76	v
119	167	77	w
120	170	78	x
121	171	79	y
122	172	7A	z
123	173	7B	{
124	174	7C	|
125	175	7D	}
126	176	7E	~ (TILDE)
127	177	7F	DEL (RUBOUT)

SD-05495

Figure B.1 ASCII Character codes

Appendix C
Powers of 2 Table

2^n	n	$1/2^n$
1	0	1.0
2	1	0.5
4	2	0.25
8	3	0.125
16	4	0.062 5
32	5	0.031 25
64	6	0.015 625
128	7	0.007 812 5
256	8	0.003 906 25
512	9	0.001 953 125
1 024	10	0.000 976 562 5
2 048	11	0.000 488 281 25
4 096	12	0.000 244 140 625
8 192	13	0.000 122 070 312 5
16 384	14	0.000 061 035 156 25
32 768	15	0.000 030 517 578 125
65 536	16	0.000 015 258 789 062 5
131 072	17	0.000 007 629 394 531 25
262 144	18	0.000 003 814 697 265 625
524 288	19	0.000 001 907 348 632 812 5
1 048 576	20	0.000 000 953 674 316 406 25
2 097 152	21	0.000 000 476 837 158 203 125
4 194 304	22	0.000 000 238 418 579 101 562 5
8 388 608	23	0.000 000 119 209 289 550 781 25
16 777 216	24	0.000 000 059 604 644 775 390 625
33 554 432	25	0.000 000 029 802 322 387 695 312 5
67 108 864	26	0.000 000 014 901 161 193 847 656 25
134 217 728	27	0.000 000 007 450 580 596 923 828 125
268 435 456	28	0.000 000 003 725 290 298 461 914 062 5
536 870 912	29	0.000 000 001 862 645 149 230 957 031 25
1 073 741 824	30	0.000 000 000 931 322 574 615 478 515 625
2 147 483 648	31	0.000 000 000 465 661 287 307 739 257 812 5
4 294 967 296	32	0.000 000 000 232 830 643 653 869 628 906 25
8 589 934 592	33	0.000 000 000 116 415 321 826 934 814 453 125

Figure C.1 Powers of 2 Table

Appendix D
Fault Codes

The following tables contain an explanation of the fault codes returned in AC1 for protection, stack, and decimal/ASCII faults.

Protection Faults

Table D.1 lists the protection violation fault codes and the interpretations.

AC1 Code (octal)	Meaning
0	Read violation
1	Write violation
2	Execute violation
3	Validity bit protection (SBR or PTE)
4	Inward address reference
5	Defer (indirect) violation
6	Illegal gate -- out of bounds or bracket compare
7	Outward call
10	Inward return
11	Privileged instruction violation
12	I/O protection violation
14	Invalid microinterrupt return block
15	Unimplemented instruction

Table D.1 Protection Violation Fault Codes

Stack Faults

Table D.2 lists the wide stack fault codes and their meanings. The processor does not return a fault code for a narrow stack fault.

AC1 Code	Meaning
000000	Overflow on every stack operation other than SAVE, WMSP, or segment crossing.
000001	Underflow or overflow would occur if the instruction were executed — WMSP, WSSVR, WSSVS, WSAVR, WSAVS. (PC in return block references the instruction that caused the stack fault.)
000002	Too many arguments on a cross segment call.
000003	Stack underflow.
000004	Overflow due to a return block pushed as a result of a microinterrupt or fault.

Table D.2 Wide Stack Fault Codes

Decimal/ASCII Faults

Table D.3 lists the decimal and ASCII fault codes. The first and second columns list the code that appears in AC1. The third column lists the instruction that caused the fault. The last column describes the conditions that can cause the fault.

Code Returned in AC1		Faulting Instruction	Meaning
Narrow	Wide		
000000	100000	**EDIT, WEDIT**	An invalid digit or alphabetic character encountered during execution of one of the following subopcodes: DMVA, DMVF, DMVN, DMVO, DMVS
000001	100001	**LDIX, STIX** **EDIT, WEDIT WLDIX, WSTIX, WDMOV, WDCMP, WDINC, WDDEC**	Invalid data type (7) Invalid data type (6 or 7)
000002	100002	**EDIT, WEDIT**	DMVA or DMVC subopcode with source data type 5; AC2 contains the data size and precision
000003	100003	**EDIT, WEDIT**	An invalid opcode; AC2 contains the data size and precision
000004	100004	**STI, LDI, WSTI, WLDI** **STIX, LDIX, WSTIX, WLDIX**	Number too large to convert to specified data type . number $> (10^{16}) - 1$ Number too large to convert to specified data type. Number $> (10^{32}) - 1$
000006	100006	**WLSN, WLDI, LSN, LDI LDIX, WLDIX** **EDIT, WEDIT, WDMOV, WDCMP, WDINC, WDDEC**	Sign code is invalid for this data type
000007	100007	**WLSN, WLDI, WLDIX, LSN LDI, LDIX, WDMOV, WDCMP, WDINC, WDDEC**	Invalid digit

Table D.3 Decimal and ASCII fault codes

Appendix E
Glossary

Some readers may be unfamiliar with the terms used to describe the features of the 32-bit processor, so the following section provides a brief definition of terms.

The Addressing Scheme

Logical Addresses

The computer uses 31-bit word addresses and 32-bit byte addresses, which can refer to all 4 Gbytes of the logical address space.

Segmentation

The large logical address space is divided, or segmented, into eight smaller logical address spaces. Each of these eight *segments* is a complete address space of 512 Mbytes.

Mapping and Demand Paging

The size of the logical address space means that not all logical locations can be represented in physical memory at the same time. The *demand paging* system moves pages between physical memory and a storage device upon demand and also keeps track of pages currently in memory. The *address translator* translates the specified logical address to its physical equivalent.

Page

A page is a 2-Kbyte block of contiguous logical addresses. The demand paging system uses the page as the smallest unit of logical memory that can be moved between physical memory and storage devices.

Page Table

A page table is made up of *page table entries (PTEs)*. Each PTE contains information about one page. The processor uses this information when translating a logical address to a physical one. A page table contains up to 512 PTEs.

Protection

The system uses a hardware-implemented hierarchical protection system that allows programs different levels of privilege. Each segment has a different level, or *ring*, of protection associated with it. This means that each ring governs the associated segment with a different degree of privilege. Ring 0 has the highest degree of protection; thus, the kernel of the operating system resides in segment 0.

The Instruction Set

The instruction set is a superset of the previous (16-bit) ECLIPSE instruction set. In this manual, the new 32-bit instructions are referred to as *wide* instructions. The 16-bit instructions supported by the 32-bit processor, but which are also supported by previous (16-bit) ECLIPSE computers (such as the ECLIPSE C/350 computer), are referred to as C/350 instructions.

Wide Instructions

These instructions manipulate data with lengths of 8, 16, or 32 bits. The mnemonics of the instructions indicate the size of the data fields referred to. The mnemonic preceded by the letter N manipulates 16-bit (narrow) data; W, 32-bit (wide) data. There is no special mnemonic prefix for those instructions that manipulate 8-bit data.

There are also mnemonic prefixes that indicate the addressing range of the instruction. X indicates that the instruction has a 512-Mbyte (extended) offset addressing range: L, a 4-Gbyte (long) addressing range.

C/350 Compatibility

The 32-bit processor supports the instruction mnemonics and binary opcodes of most instructions implemented on the ECLIPSE C/350 computer. This means that most programs that execute on the C/350 will also execute on the 32-bit processor without recompiling or reassembling.

Note that the C/350 instructions maintain their limitations of the lower 64-Kbyte addressing range.

True and Impure Zero

Floating-point zero is represented by a number with all bits zero, known as *true zero.* If a number has a zero mantissa but not a zero sign or exponent, it is called *impure zero.* When representing zero as a floating-point number, use true zero; impure zero produces undefined results in calculations.

Normalized Format

A nonzero mantissa represents a fraction from 1/16 to $1-2^{-56}$. A floating-point number represented in this way is said to be *normalized*. Note that impure zero is not in normalized form. Most floating-point instructions require normalized operands if they are to produce correct results. Floating-point numbers that are not normalized or are not true zero produce undefined results except as noted.

Magnitude

The magnitude of a floating-point number is defined as follows:

$Mantissa\ X\ 16^{y}$

where y is the true value of the exponent.

Index

Within the index, the letter "f" following a page entry indicates "and the following page"; the letters "ff" following a page entry indicate "and the following pages".

15-bit displacement 1-11
16-bit data, edit decimal and alphanumeric 9-8, 10-35f
16-bit fixed-point logical format 2-12
16-bit fixed-point two's complement 2-2
16-bit instructions, executing 9-1
16-bit integer, convert to 10-19
16-bit programs, supporting 4-1
28-bit displacement 1-11
31 bits, displacement sign-extending to 1-11
32-bit fixed-point arithmetic, expanding to 9-3f
32-bit fixed-point logical format 2-12
32-bit fixed-point two's complement 2-2
32-bit processor instruction, equivalent 9-7
32-bit programs, supporting 4-1
32-bit subroutine, calling C/350 subroutine from a 9-4
32-bits to 16-bits, convert from 2-2
32-bits, sign or zero extend 16-bits to 2-2
6-bit device address 7-4
64 Kbytes of segment, first 9-2
64 Kbytes, expanding beyond 9-3
8-bit displacement 1-11
@ field 1-15
@ flag 9-5f

A

Aborting floating-point division 3-13
Aborting instruction execution 2-24, 8-2
Aborting memory reference instruction 8-2
Aborting nonprivileged fault 5-14
Aborting subroutine call 5-11f
Absolute address 1-10, 1-12f
Absolute value 3-3, 10-42
AC (see accumulator)
AC0 with address of fault, loading 5-25
AC0,
  load PSR in 2-9f, 10-83
  store PSR from 2-9, 10-111
AC1 with fault code, loading 5-24f, 9-3
AC1, fault code in 2-23, 5-21, 5-23ff, 9-3, D-2
AC2 relative address mode 1-12
AC2 with base address of DCT, loading 7-12
AC3 relative address mode 1-12
Access fault, privileged 5-12
Access flag,
  execute 8-6
  read 8-6
  valid page 8-5
  write 8-6
Access instructions, double-word stack 4-5
Access privileges destination segment 5-12
Access request 8-8
Access restrictions, memory 1-6
Access validation 8-8
Access violation,
  execute 5-15
  read 5-15
  write 5-15
Access,
  device 7-1
  execute 1-9
  ignoring page protocol 8-7f
  operand 1-12
  page 8-5, 8-8
  read 1-9
  segment 8-2
  type of page 8-5
  type of segment 8-2
  valid page 8-5
  valid read 8-6
  valid segment 8-2
  validate I/O 8-2
  write 1-9
Accessing destination segment 5-9f

Accessing device control table 7-12
Accessing double word in a stack 4-4
Accessing memory 1-8, 8-2
Accessing page table 8-5
Accessing page to read, write, or execute 8-5
Accessing physical page 8-2
Accessing protection mechanism 1-17
Accumulator (also see AC0, AC1, etc.)
Accumulator bit instructions, wide skip on 2-23
Accumulator bit set to one, wide skip on 2-8, 10-168
Accumulator bit set to zero, wide skip on 2-8, 10-168
Accumulator contents, executing 5-2
Accumulator equal to immediate, wide skip if 2-23, 10-166
Accumulator greater than immediate,
  wide skip if 2-8, 10-167
  wide unsigned skip if 2-8, 10-174
Accumulator in double word addressed by WSP, store 2-3, 4-5
Accumulator in stack pointer contents, store 10-113
Accumulator in WFP, store 4-4, 10-112
Accumulator in WSB, store 4-4, 10-112
Accumulator in WSL, store 4-4, 10-112
Accumulator in WSP, store 4-4, 10-112
Accumulator less than or equal to immediate,
  wide skip if 2-8, 10-169
  wide unsigned skip if 2-8, 10-175
Accumulator not equal to immediate, wide skip if 2-8, 10-170
Accumulator relative addressing, C/350 9-1
Accumulator to memory word, narrow add 2-4
Accumulator with double word addressed by WSP, load 4-5
Accumulator with double word, load 10-68
Accumulator with WFP, load 4-4, 10-67
Accumulator with WSB, load 4-4, 10-68
Accumulator with WSL, load 4-4, 10-68
Accumulator with WSP, load 4-4, 10-68
Accumulator,
  C/350 fixed-point 9-1
  C/350 floating-point 9-1
  double word addressed by WSP load 2-3
  extended load 9-8, 10-38
  extended store 9-8, 10-41
  fix to 10-45
  fixed-point 1-2
  float from 10-47
  load 9-8, 10-67
  loading byte in 2-20
  loading physical address into 7-3
  narrow add memory word to 2-4, 10-78, 10-184
  narrow load 2-3, 9-8, 10-80, 10-186
  narrow skip on all bits set in 2-8, 10-98
  narrow skip on any bit set in 2-8, 10-99
  narrow store 2-3, 9-8, 10-80, 10-187
  store 9-8, 10-111
  storing byte address in 2-23
  transfers between device and 7-1
  wide add memory word to 2-4, 10-86, 10-191
  wide fix from floating-point 3-3f, 3-13, 9-9, 10-141
  wide float from 3-3f
  wide float from fixed-point 3-3f, 9-9, 10-141
  wide load 2-3, 10-88, 10-193
  wide skip on all bits set in 2-8, 10-163
  wide skip on any bit set in 2-8, 10-164
  wide store 2-3, 10-89, 10-193
Accumulators,
  exchange 2-3, 10-178
  initializing 5-15f
  pop floating-point status and 10-51f
  pop multiple 9-8, 9-11, 10-102, 10-160
  popping 4-1
  push floating-point status and 10-52f
  push multiple 9-8, 9-11, 10-103f
  pushing 4-1
  restoring 5-4
  storing 5-7
  wide pop multiple 2-3, 4-1, 4-5, 4-8, 9-8, 9-11, 10-160
  wide push 2-3, 4-5, 4-8, 9-8, 10-161
Accuracy of result, increasing 3-4
ADC 2-4, 2-6, 2-8, 10-2f
ADD 2-4, 2-6, 2-8, 10-3ff
Add accumulator to memory word, narrow 2-4
Add and move, block 9-8, 10-8f
Add complement,
  wide 2-4, 10-118
  with optional skip 2-4, 2-6, 2-8, 10-2f
Add double (FPAC to FPAC) 3-6, 10-42
Add double (memory to FPAC) 3-6, 9-9, 10-42, 10-72, 10-179
Add immediate 2-4, 10-5f
Add immediate,
  extended 2-4, 10-5
  narrow 2-4, 10-78, 10-93, 10-185
  narrow extended 2-4, 10-92
  wide 2-4, 10-86, 10-119, 10-191
Add instructions, fixed-point 2-22
Add memory word to accumulator,
  narrow 2-4, 10-78, 10-184
  wide 2-4, 10-86, 10-191
Add single (FPAC to FPAC) 3-6, 10-43
Add single (memory to FPAC) 3-6, 9-9, 10-42f, 10-72, 10-180
Add to DI 10-20
Add to P 10-21
Add to P depending on S 10-20
Add to P depending on T 10-21
Add to SI 10-21
Add with narrow immediate, wide 2-4, 10-159

Add with optional skip 2-4, 2-6, 2-8, 10-3ff
Add with wide immediate, wide 2-4, 10-119
Add,
  decimal 10-20
  narrow 2-4, 10-92
  wide 2-4, 10-119
  with optional skip 2-6
ADDI 2-4, 10-5
Adding one to the intermediate mantissa 3-6
Addition instructions,
  fixed-point 2-4
  floating-point 3-6
Addition, floating-point 3-5f
Additional bytes, inserting 2-21
Address boundary, two Kbyte 8-3
Address causing fault, decimal instruction 5-23
Address field,
  device 7-4
  word 1-14f
Address in accumulator, storing byte 2-23
Address into accumulator, loading physical 7-3
Address mnemonic, device 7-4
Address mode, AC and PC relative 1-12
Address of block to transfer, starting 7-2
Address of DCT, loading AC2 with base 7-12
Address of decimal fault instruction 5-24
Address of fault instruction,
  at time of fault 5-14ff
  determining 5-21
Address of fault, loading AC0 with 5-25
Address of first entry in page table 8-2ff
Address of first word for I/O transfer 7-2
Address of floating-point instruction 3-14
Address of sub-block to transfer, starting 7-2
Address of WEDIT fault instruction 5-23
Address parameter, vector stack fault 7-9
Address ranges 1-10, 1-12, 8-7, 9-10
Address reference fault code, inward D-1
Address reference protection violation, inward 5-17
Address space,
  C/350 9-4ff
  logical 8-1, 8-11
  memory 1-1
  valid 2-22
Address to device, sending physical 7-2
Address translation,
  device 7-1, 7-6
  memory 8-2f, 8-7
Address translator, purging 8-12, 10-101
Address wraparound 1-4f, 5-1, A-2
Address,
  6-bit device 7-4
  absolute 1-12
  base 1-11, 7-12
  C/350 effective 9-2, 9-5
  calculating an effective 2-21f
  checking range of the logical 8-7
  direct 1-15
  effective 1-10f, 1-16, 2-20f, 5-2, 5-4f, 5-10, 7-6, 7-11, 8-2
  effective logical 8-7
  effective relative 1-11
  extended load effective 9-10, 10-39
  forming byte 9-5f
  forming physical 7-2
  indirect 1-10, 1-15
  indirect logical 8-7
  indirect relative 1-11
  intermediate logical 1-11
  load effective 2-12ff, 2-21, 8-3, 9-10, 10-71, 10-77, 10-184
  load effective byte 1-13, 2-21, 10-77, 10-184
  load physical 7-2
  logical 5-1, 8-2f, 8-5, E-1
  logical address to physical 8-1
  most significant bits translating logical 7-2
  narrow return 5-27
  one Mbyte or less logical 8-3
  one Mbyte to 512 Mbytes logical 8-3
  one-level page table logical word 8-7
  PC return 4-6
  physical 1-7, 1-11, 7-2, 8-1ff
  physical page 8-7
  push 2-21, 4-5, 4-8, 10-81, 10-189
  push byte 2-21
  push effective 2-21
  push effective byte 2-20f, 4-5f, 4-8, 10-81, 10-189
  push return 5-6, 9-10f, 10-104
  return 5-2, 5-7, 5-13, 5-19f, 5-25
  shifted page 8-4
  skip on valid byte or word 2-22
  store return 5-7, 7-7
  translating logical 1-7, 7-2f, 8-2, 8-5
  two-level page table logical word 8-7
Addressing another page table page 8-7
Addressing another segment 1-9
Addressing chains, indirect 1-11
Addressing memory for I/O data transfer 7-2
Addressing next instruction 5-1
Addressing protection violation, indirect 1-11
Addressing range 1-10, 8-3, 8-7
Addressing scheme E-1
Addressing violation, indirect 5-15
Addressing,
  absolute 1-10, 1-13
  byte and word formats for 1-9
  C/350 9-1ff
  device control table 7-12
  direct 8-7
  effective 1-13
  entry in the vector table 7-12

indirect 1-8, 1-11, 8-7, 9-5f
mapped memory 1-5
memory 1-5, 1-8
n variable in PC relative 1-11
relative 1-10f, 1-13
unmapped memory 1-5
ADI 2-4, 10-5f
Algorithms, floating-point numerical A-2
Aligning mantissa 3-4f
All bits set in accumulator,
narrow skip on 10-98
wide skip on 10-163
All bits set in memory location,
narrow skip on 10-99
wide skip on 10-163
Alphabetical characters, move j 2-21
Alphabetics, move 10-27
Alphanumeric 16-bit data, edit decimal and 9-8, 10-35f
Alphanumeric, wide edit of 5-5
Altering normal program flow 5-1
Always, skip 10-54
ANC 2-13, 10-6
AND 2-6, 2-12ff, 10-6ff
AND immediate 2-13, 10-8
AND immediate, wide 2-13, 10-120
AND with complemented source 2-13, 10-6, 10-119
AND with optional skip 2-6, 2-12ff, 10-6ff
AND, wide 2-13, 10-120
ANDI 2-13, 10-8
Anomalies A-1
Another segment 1-9, 4-2, 4-4, 5-6
ANY flag 3-12f, 5-20
Appending guard digit 3-4
Arccosine double, floating-point 10-135
Arccosine instruction error, floating-point 3-14
Arccosine single, floating-point 10-135f
Arccosine, floating-point 3-11
Arcsine double, floating-point 10-136
Arcsine instruction error, floating-point 3-14
Arcsine single, floating-point 10-136f
Arcsine, floating-point 3-11
Arctangent (two-accumulator), floating-point 3-11
Arctangent double (two-accumulator), floating-point 10-138
Arctangent double, floating-point 10-137
Arctangent single (two-accumulator), floating-point 10-138
Arctangent single, floating-point 10-137
Arctangent, floating-point 3-11
Argument (INV = 0), skip on no invalid input 10-57
Argument count 5-7
Argument error flag, floating-point input 3-13
Arguments,
save without 9-10f, 10-107f
transferring call 5-10ff
Arithmetic example, decimal 2-25
Arithmetic instructions,
decimal 2-22
decimal string 2-22
fixed-point 2-4
floating-point 3-6
Arithmetic shift instructions, wide 2-6
Arithmetic shift with narrow immediate, wide 10-121
Arithmetic shift, wide 10-120
Arithmetic,
binary coded decimal 2-1
expanding to 32-bit fixed-point 9-3f
Array,
expanding data areas for large 9-3
gate 5-10f
ASCII characters 2-15, 2-17
ASCII codes B-1
ASCII data fault D-1
ASCII data fault code D-2
ASCII data,
fault 5-21ff
invalid format 5-13
type 2 or 3 fault return block for 5-23
valid type 2-24
ASCII fault code 2-24, 5-22
ASCII fault codes, decimal and 2-23
ASCII or decimal fault code 2-24, 5-14, 5-21f, D-2
Assignments and format, map register 7-2
Attempting division by zero 2-8, 5-19f

B

Backward link 6-2
Backward search queue and skip,
narrow 5-3, 6-5, 10-93f
wide 5-3, 6-5, 10-122f
BAM 9-8, 10-8f
Base address 1-11, 7-12
Base level interrupt processing 5-8, 7-9ff, 10-162
Base of vector table 7-12
Base register, segment 5-11, 8-1f, 8-7
Base,
initializing wide stack 4-2, 8-11
loading wide stack 5-11, 5-13
wide stack 4-2f, 4-7, 5-11, 5-13, 5-25, 7-9, 7-11, 8-11
BCD 2-1, 2-15, 2-17, 2-22
BI field 1-14
Binary coded decimal arithmetic 2-1
Binary conversion instructions, floating-point 3-3
Binary logarithm double, floating-point 10-141f
Binary logarithm single, floating-point 10-142
Binary logarithm, floating-point 3-11
Binary operations 2-1
Binary point location 3-2

Bit addressing,
format C/350 9-6f
format wide 1-15
Bit and set bit to one,
skip on zero 9-6, 9-8, 10-117
wide skip on zero 9-8, 10-174
Bit identifier 1-15, 9-6f
Bit operand 1-15
Bit pointer 1-15
Bit set in accumulator,
narrow skip on any 10-99
wide skip on any 10-164
Bit set in memory location,
narrow skip on any 10-99
wide skip on any 10-164
Bit set to one, wide skip on accumulator 10-168
Bit set to zero, wide skip on accumulator 10-168
Bit three of SBR, setting 7-2
Bit to one,
set 9-6f, 10-10
skip on zero bit and set 9-6, 9-8, 10-117
wide set 9-8, 10-123f
wide skip on zero bit and set 9-8, 10-174
Bit to zero,
set 9-6f, 10-10f
wide set 9-8, 10-124
Bit two of SBR, setting 7-1ff
Bit zero of fault code in AC1 9-3
Bit zero of narrow stack limit 5-27f
Bit zero of narrow stack pointer 5-27f
Bit zero of segment base register 8-2
Bit zero of wide stack limit 5-25f
Bit zero of wide stack pointer 5-25f
Bit,
checking validity 5-11
indirect 9-5f
locate lead 10-81
mask 7-12
sign 3-2
skip on nonzero 9-6ff, 10-110
skip on zero 9-8
wide locate and reset lead 10-155
wide locate lead 10-155
wide skip on zero 9-8, 10-174
Bits set in accumulator,
narrow skip on all 10-98
wide skip on all 10-163
Bits set in memory location,
narrow skip on all 10-99
wide skip on all 10-163
Bits,
count 9-8, 10-16
displacement (sign-extending to 31) 1-11
ignoring page access protocol 8-7
INP 5-20
load modified and referenced 8-12, 10-77
OR referenced 8-12, 10-100f
reset referenced 8-12, 10-104f
store modified and referenced 8-12, 10-110
table of referenced and modified 8-1
wide count 9-8, 10-128
BKPT 2-9, 4-5, 4-8, 5-5f, 10-9
BLM 9-8, 10-10
Block add and move 9-8, 10-8f
Block and execute,
pop 2-9, 4-5, 4-8, 5-5f, 5-28, 9-10f, 10-101, 10-103
wide pop 2-9, 4-5, 4-8, 5-8, 5-14, 5-26, 9-10f, 10-160f
Block in a stack, accessing double word 4-4
Block instructions, wide stack return 4-5
Block move 9-8, 10-10
Block move, wide 2-3, 9-8, 10-121f
Block to transfer, starting address of 7-2
Block,
C/350 return 9-2
data 4-4
fault return 5-15f
floating-point fault return 5-20
narrow return 9-2
pop 5-28, 9-10f, 10-103
popping return 4-1, 4-5
popping wide return 4-3
pushing return 4-1, 4-5, 9-3
return 4-4, 9-10f, D-2
stack fault return 4-8, 5-26, 5-28
standard wide return 4-6
wide pop 2-9, 4-5, 4-8, 5-5f, 5-8, 5-14, 5-26, 9-10f, 10-160f
wide pop context 2-9, 8-8, 8-11f, 10-133
wide return 4-3
Blocking an interrupt request 7-5f
Blocks of words, between memory and device transferring 1-6
BMC device 1-5
BMC maps 7-3
Boundary,
byte 1-2
two Kbyte address 8-3
word 1-2, 3-2
Bracket 5-10ff
Branch (PC relative jump) 5-2
Branch, wide 9-10, 10-122
Breakpoint 10-9
Breakpoint handler 2-9, 4-5, 4-8, 5-5f
BTO 9-6ff, 10-10
BTZ 9-6ff, 10-10f
Building a queue 6-1
Building device control table 7-12
Burst multiplexor channel 1-5f, 7-1f
BUSY flag 7-3, 7-5
Byte address,
load effective 1-13, 2-21, 10-77, 10-184

push 2-21
push effective 2-20f, 4-5f, 4-8, 10-81, 10-189
Byte addressing format 1-9f, 9-5f
Byte manipulation 2-1
Byte pointer 1-13f, 5-12, 9-5f
Byte pointer to fault subopcode 5-24
Byte pointer to subopcode causing fault 5-23
Byte pointer to word pointer, converting 2-21
Byte pointer, skip on valid 10-117
Byte,
BCD digits per 2-17
boundary 1-2
compare instructions 2-15
extended load 9-8, 10-38
extended store 9-8, 10-41
indicator 1-13, 9-5f
load 2-20, 9-8, 10-69, 10-77, 10-184
move instructions 2-20
operand 1-13
operations 2-15
packed 2-15
skip on valid 2-22
store 2-20, 2-23, 9-8, 10-86, 10-113, 10-190
unpacked 2-15
wide load 2-20, 9-8, 10-153
wide store 2-20, 9-8, 10-172
Bytes in memory, moving 2-15
Bytes,
deleting 2-15
inserting 2-15, 2-21f
moving 2-20
swapping two 2-6

C

C/350 accumulator relative addressing 9-1
C/350 address space 9-4ff
C/350 bit addressing, format 9-6f
C/350 byte addressing 9-5f
C/350 commercial faults A-3
C/350 compatibility E-2
C/350 decimal and ASCII fault handler 9-3
C/350 effective address 9-2, 9-5
C/350 equivalent instruction 9-8, 9-11
C/350 faults and interrupts 9-3
C/350 fixed-point accumulator 9-1
C/350 fixed-point instructions 9-7f
C/350 floating-point accumulator 9-1
C/350 floating-point fault handler 9-3
C/350 floating-point instructions 9-9
C/350 instruction results A-1
C/350 instructions 9-4
C/350 memory reference instruction 9-1, 9-4ff
C/350 pointers 9-3
C/350 program counter 9-2
C/350 program development 9-2ff
C/350 program flow instructions 9-2, 9-10
C/350 registers 9-1
C/350 return block 9-2
C/350 signed divided instructions A-2
C/350 stack 9-2
C/350 stack instructions 9-11
C/350 stack-referenced instructions 9-1
C/350 word addressing, format 9-5
Calculating a two's complement number 2-8, 5-19
Calculating a two's complement number, out of range 5-20
Calculating an effective address 2-21f
Calculating the result 3-5
Call arguments, transferring 5-10ff
Call fault code, outward D-1
Call protection violation, outward 5-17
Call subroutine 2-9, 3-13, 4-1, 4-3, 4-5, 4-8, 5-5f, 5-8, 5-13, 10-66f, 10-177f
Call,
aborting subroutine 5-11f
executing subroutine 5-1
illegal outward subroutine 5-9
inward 5-8
returning from 4-1, 4-5
subroutine 5-4
too many arguments segment 5-26, D-2
Calling C/350 subroutine from a 32-bit subroutine 9-4
Calling sequence 5-13
Calls, I/O system 7-5
Capabilities, functional and system protection 1-1
Capacity, exceeding processor storage 1-2
CARRY flag 2-5f, 4-6, 5-6f, 5-19f, 5-23, 5-26, 9-2
Carry to one or zero, set 10-18
Carry,
complement 10-18
decimal 2-22, 10-31f
floating-point 3-6
restoring 5-4
CARRY, skip on (see ADC, ADD, AND, COM, INC, MOV, NEG, or SUB instruction)
Carry, storing 5-7
Cause of page fault 8-11
Causing fault,
byte pointer to subopcode 5-23
decimal instruction address 5-23
instruction 2-24, 5-22
Central processor identification instructions 8-12
Chain,
indirect addressing 1-11
pointer 7-13
Change, sign bit 2-6
Changing a device flag 7-4
Changing C/350 subroutine 9-4
Changing current segment of execution 7-6

Changing interrupt mask 7-6
Changing protection mechanism 1-17
Changing RND flag 3-4
Channel I/O, burst multiplexor and data 1-5f, 7-1ff
Channel mask, I/O 7-12
Character compare 9-8, 10-12f
Character compare, wide 2-23, 9-8, 10-125f
Character depending on sign flag, insert 2-21
Character depending on trigger, insert 2-21
Character j times, insert 2-21, 10-25
Character move 5-12, 9-8, 10-15f
Character move until true 9-8, 10-13f
Character move until true, wide 2-20, 9-8, 10-126f
Character move, wide 2-20, 9-8, 10-127f
Character once, insert 2-21, 10-25
Character scan until true, wide 2-23, 10-128f
Character suppress, insert 10-25
Character translate and compare 9-8, 10-18f
Character translate and compare, wide 2-20, 2-23, 9-8, 10-129f
Character, ASCII 2-17
Characters immediate, insert 2-21, 10-25
Characters,
  inserting or converting string of 2-20
  move 10-28
  move j 2-21
Check, valid segment 8-3
Checking data transfers 5-12
Checking for stack overflow 5-14
Checking for stack overflow fault 4-3
Checking for valid data and type 5-13, 5-21ff
Checking for valid operations 5-13
Checking for vector stack overflow 7-13
Checking for wide stack overflow fault 4-3
Checking range of the logical address 8-7
Checking validity bit 5-11
CIO 7-3, 10-11
CIOI 7-3, 10-11
Clearing FPSR errors 3-12, 10-43
CLM 9-8, 10-12
CMP 9-8, 10-12f
CMT 9-8, 10-13f
CMV 9-8, 10-15f
COB 9-8, 10-16
Code in AC1, fault 2-23, 5-21, 5-23ff, 9-3, D-2
Code,
  ASCII fault 2-24, 5-22
  ASCII/decimal fault 2-24, 5-22
  decimal fault 2-24, 5-22
  floating-point identification 3-13
  ID 3-13
  loading AC1 with fault 5-24f, 9-3
  protection fault error 8-8
  violation fault D-1
  wide stack fault 5-26, D-1f
Codes,
  ASCII B-1
  decimal or ASCII fault D-2
  fault D-1f
  violation fault D-1
COM 2-6, 2-13f, 10-16f
Command I/O 7-3, 10-11
Command I/O immediate 7-3, 10-11
Command, issuing programmed I/O 7-3
Commercial faults, C/350 A-3
Common logarithm double, floating-point 10-143
Common logarithm single, floating-point 10-144
Common logarithm, floating-point 3-11
Communicating with I/O controller 7-3
Communication, interrupt 7-5
Compare instructions, byte 2-15
Compare to limits and skip 9-8, 10-12
Compare to limits and skip, wide 2-8, 9-8, 10-124
Compare two floating-point numbers (set N and Z) 3-8f, 10-43
Compare,
  character 9-8, 10-12f
  character translate and 9-8, 10-18f
  wide character 2-23, 9-8, 10-125f
  wide character translate and 2-20, 2-23, 9-8, 10-129f
  wide decimal 2-22, 10-130f
Comparing data types 2-15
Comparing divisor and dividend 3-8
Comparing DO-loop variable to constant 5-3
Comparing the gate number 5-11
Comparing wide stack pointer 4-3
Compatibility,
  C/350 E-2
  upward program 4-1, 9-1
Complement 2-13, 10-16f
Complement CARRY flag 2-6, 10-18
Complement number, out of range 5-19f
Complement,
  add 2-4, 2-6, 10-2f
  wide 10-128
  wide add 10-118
Complemented source, AND with 10-6, 10-119
Complementing the mantissa sign 3-7
Computation,
  fixed-point 1-2, 1-16, 2-1
  floating-point 1-3, 1-16, 2-15, 3-1
Computing instructions, C/350 9-8f
Computing narrow data 2-2
Concluding vector interrupt service 7-11
Condition,
  stack overflow or underflow 4-3
  testing BUSY or DONE flag and skip on 7-3
  testing machine 5-2
Conditional skip, load physical and 10-82
Constant, comparing DO-loop variable to 5-3
Constants, storing device dependent 7-12

Constructing double word for vector stack 7-12
Contents,
  executing accumulator 5-2
  modifying stack register 4-2
  reserved memory 7-11
Context block, wide pop 2-9, 8-8, 8-11f, 10-133
Control, transferring program 1-8, 4-2, 4-4, 7-13
Controller, device 7-3ff
Controlling ION flag 7-6
Conversion instructions, floating-point 3-3
Conversion,
  fixed-point precision 2-2
  fixed-point to floating-point 2-20
  floating-point to fixed-point 2-20, 3-3f, A-1
Convert decimal/floating-point instructions 2-20
Convert FPAC data and load in memory 2-20
Convert from 32-bits to 16-bits 2-2
Convert instructions, decimal move and 2-20
Convert the four FPAC's and load in memory 2-20
Convert to 16-bit integer 10-19
Convert, number too large to D-2
Converting a decimal and loading in FPAC 2-20
Converting a decimal integer 2-21
Converting byte pointer to word pointer 2-21
Converting data types 2-15
Converting decimal numbers 2-15
Converting decimal to floating-point 3-3
Converting mixed number to fraction 3-3
Converting packed decimal data 2-20
Converting string of characters 2-20
Converting to double or single precision 3-1
Converting to wide data 2-2
Copying arguments from the source stack 5-12
Cosine double, floating-point 10-139
Cosine single, floating-point 10-139
Cosine, floating-point 3-11
Count bits 9-8, 10-16
Count bits, wide 2-13, 9-8, 10-128
Count,
  argument 5-7
  LCALL or XCALL 4-6
Counter wraparound, program A-1
Counter,
  floating-point program 3-8, 3-10
  incrementing program 3-9
  program 2-7, 2-22, 7-13
CPU device 7-4ff
CPU identification,
  C/350 load 8-12, 10-35
  long load 8-12, 10-67
  narrow load 8-12, 10-94
Crossing to segment zero 7-9, 7-11, 8-11
Crossing violation, segment 5-15
Crossing, segment 5-6, 5-8, 5-26, D-2
CRYTC 2-6, 10-18
CRYTO 2-6, 10-18
CRYTZ 2-6, 10-18
CTR 9-8, 10-18f
Current interrupt mask 7-12f
Current segment 1-8, 4-1, 5-1, 5-6f, 5-13, 5-25, 5-28, 7-9, 7-11, 8-11, 9-2, 9-4ff
Current segment field, modifying 1-5
Current segment of execution, changing 7-6
Current segment, defined 1-4
Current state of processor, saving 8-8, 8-11
Current value of P 5-23f
CVWN 2-2, 10-19

D

DAD 2-22, 10-20
DADI 2-21, 10-20
DAPS 2-21, 10-20
DAPT 2-21, 10-21
DAPU 2-21, 10-21
DASI 2-21, 10-21
Data and type, checking for valid 5-13, 5-21ff
Data areas, expanding 9-3
Data block 4-4
Data channel I/O 1-5f, 7-1ff
Data element 6-1
Data element,
  dequeuing 6-1, 6-5, 10-22
  enqueuing 6-1, 6-4, 10-39f
Data format 2-15
Data format,
  fixed-point logical 2-12
  invalid ASCII or decimal 5-13
Data from the wide stack, retrieving 4-3
Data in A (to A buffer of device) 7-3, 10-24
Data in area beyond the stack, overwriting 5-25
Data in B (to B buffer of device) 7-3, 10-24
Data in C (to C buffer of device) 7-3, 10-24
Data in the stack, storing 4-2
Data instructions, wide stack 4-4
Data move instructions, floating-point 3-4
Data out A (from A buffer of device) 7-3, 10-30
Data out B (from B buffer of device) 7-3, 10-31
Data out C (from C buffer of device) 7-3, 10-31
Data structure, implicit 2-15
Data transfer,
  checking 5-12
  formatted 2-20
  I/O 1-5, 7-1ff
  requesting I/O 7-5
Data type 6 or 7, invalid 2-24, 5-22, D-2
Data type description, explicit 2-15
Data type faults 2-24
Data type indicator, explicit 2-16
Data type,
  explicit 2-24
  invalid sign code for 2-24, 5-22
  valid ASCII or decimal 2-24

Data types 2-15, 2-17
Data,
  accessing page to read or write 8-5
  computing narrow 2-2
  converting floating-point 2-20
  converting packed decimal 2-20
  converting to wide 2-2
  edit decimal and alphanumeric 16-bit 9-8, 10-35f
  invalid decimal 1-16
  loading floating-point 3-4
  moving bytes of 2-20
  moving floating-point 3-4
  normalizing floating-point 3-1
  sign magnitude 3-1
  storing floating-point 3-4
  transferring I/O 7-2
  type 1 fault return block for decimal 5-23f
  type 2 or 3 fault return block for ASCII 5-23
  type of 2-16
DCH 1-5
DCT 7-12
DDTK 2-21, 10-21
Decimal add 10-20
Decimal and alphanumeric 16-bit data, edit 9-8, 10-35f
Decimal and ASCII fault codes 2-23
Decimal and ASCII fault handler, C/350 9-3
Decimal and byte operations 2-15
Decimal and loading in FPAC, converting a 2-20
Decimal arithmetic example 2-25
Decimal arithmetic instructions 2-22
Decimal arithmetic operations 2-15
Decimal arithmetic, binary coded 2-1
Decimal carry 2-22, 10-31f
Decimal compare, wide 2-22, 10-130f
Decimal conversion instructions, floating-point 3-3
Decimal data fault 5-21ff
Decimal data fault code D-1f
Decimal data format, invalid 5-13
Decimal data type, valid 2-24
Decimal data,
  converting packed 2-20
  invalid 1-16
  type 1 fault return block for 5-23f
Decimal decrement, wide 2-22, 10-131
Decimal digit and sign, representing 2-17
Decimal fault code 2-23f, 5-21f
Decimal fault instruction, address of 5-24
Decimal fixed-point subtract instructions 2-22
Decimal increment, wide 2-22, 10-131
Decimal instruction address causing fault 5-23
Decimal integer, converting a 2-21
Decimal move and convert instructions 2-20
Decimal move, wide 2-20, 10-132f
Decimal numbers 2-15, 2-20
Decimal or ASCII fault code 2-24, 5-14, 5-21f, D-1f
Decimal results, shifting 2-22
Decimal shift instructions 2-22
Decimal string arithmetic instructions 2-22
Decimal string, packed and unpacked 2-17
Decimal subtract 10-31f
Decimal to floating-point, converting 3-3
Decimal,
  fixed-point 3-3
  packed 2-15
  unpacked 2-15
Decimal/floating-point instructions, convert and move 2-20
Decrement and jump if nonzero 10-21
Decrement and skip if zero,
  C/350 9-8, 10-34
  extended 9-8, 10-36
  narrow 2-9, 9-8, 10-79, 10-186
  wide 2-9, 10-87, 10-192
Decrement double word addressed by WSP 2-9, 4-5, 10-34
Decrement word and skip, fixed-point 2-9
Decrement, wide decimal 2-22, 10-131
Decrementing an intermediate exponent 3-5
Decrementing interrupt level word 7-9
Decrementing wide stack pointer 4-3f
Decrementing word in stack 2-21
Defer (indirect) protection violation 5-17
Defer (indirect) violation fault code D-1
Defining limits of narrow stack 9-2
Defining stack location 4-2
Deleting bytes 2-15
Demand paging,
  defined 8-11
  mapping and E-1
DEND 2-21, 5-6, 10-22
Dependent information, interrupt routine 7-12
Depending on S, add to P 10-20
Depending on sign flag, insert character 2-21
Depending on T, add to P 10-21
Depending on trigger, insert character 2-21
Depth fault, page table 8-7
DEQUE 6-5, 10-22
Dequeue a queue data element 10-22
Dequeue data element 6-5
Dequeuing data element 6-1
DERR 10-23
Descriptor,
  original 5-23f
  queue 6-2
Destination indicator for STIX 5-23
Destination indicator for WSTI 5-23
Destination indicator, add integer to 2-21
Destination pointer for WDMOV, WDCMP, WDINC, or WDDEC 5-23

Destination segment 5-6f, 5-10ff
Destination segment,
  access privileges 5-12
  accessing 5-9f
  defined 1-9
Destination stack overflow 5-12
Detected error 10-23
Detecting a fault 2-10, 3-13, 5-1
Detecting an error 5-13
Detecting an I/O interrupt request 5-1
Detecting an overflow fault 2-10
Detecting nonprivileged fault 5-13
Detecting power failure 7-5
Detecting privileged fault 5-13
Detecting proper power voltage ranges 7-5
Detecting protection violation 7-2
Detecting wide stack fault 5-25
Detection,
  enabling floating-point fault 3-13
  enabling vector stack underflow or overflow 7-9
Determining address of fault instruction 5-21
Development, supporting C/350 program 9-2
Device address field 7-4
Device address translation 7-1, 7-6
Device and accumulator, transfers between 7-1
Device and memory, transfers between 7-1
Device control table 7-12f
Device controller 7-4f
Device dependent constants or variables, storing 7-12
Device driver 7-2ff
Device flags 7-3ff
Device independent operations 7-3
Device instruction, CPU 7-6
Device interrupt routine 7-12f
Device interrupt system 7-5
Device management 1-5, 7-1
Device map 7-2f, 10-154
Device number, interrupting 7-12f
Device state 7-5
Device,
  access 7-1
  BMC 1-5
  identifying unique 7-4
  idle 7-5
  internal 7-4
  sending physical address to 7-2
  starting a 7-5
  vector on interrupting 10-190
Devices,
  controlling 7-3
  device flags for general 7-5
  supporting 7-1
DHXL double hex shift left 2-22, 10-23
DHXR double hex shift right 2-22, 10-23f
DI, add to 10-20
DIA 7-3, 10-24
DIB 7-3, 10-24
DIC 7-3, 10-24
DICI 2-21, 10-25
Dictionary, instruction 10-1
Digit and sign, representing decimal 2-17
Digit with overpunch, move 2-21, 10-29
Digit,
  invalid 2-24, 5-22, D-2
  significant 2-17
Digits, BCD 2-17
DIMC 2-21, 10-25
DINC 2-21, 10-25
DINS 2-21, 10-25
DINT 2-21, 10-25
Direct address 1-15, 8-7
Direction of I/O transfer 5-8, 7-2
Disable,
  fixed-point trap 2-9, 10-61
  floating-point trap 3-12, 10-60
Disabling data channel and BMC maps 7-3
Disabling I/O interrupt recognition 7-5, 7-12
Disabling stack fault 4-7, 5-25, 5-27
Disk resident page 8-1, 8-5
Disk,
  loading referenced page from 8-11
  transferring page to 8-11
Dispatch 5-2, 9-10, 10-32f, 10-70f
Displacement 1-11
Displacement, XVCT 7-11
DIV 2-5, 10-26
Divide by zero error, floating-point 3-14
Divide double (FPAC by FPAC) 3-8, 10-44
Divide double (FPAC by memory) 3-8, 9-9, 10-44, 10-72, 10-180
Divide instruction, floating-point 3-13
Divide memory word,
  narrow 2-5, 10-78, 10-185
  wide 2-5, 10-87, 10-191
Divide single (FPAC by FPAC) 3-8, 10-45
Divide single (FPAC by memory) 3-8, 9-9, 10-44, 10-73, 10-180
Divide,
  narrow 10-94
  narrow sign extend 2-5
  sign extend and 2-5, 10-27
  signed 2-5, 10-26
  skip on no overflow and no zero (OVF and INV = 0) 3-9
  skip on no underflow and no zero (UNF and INV = 0) 3-9
  skip on no zero (INV = 0) 3-9
  unsigned 2-5, 10-26
  wide 2-5, 10-132
  wide signed 2-5, 10-132

Divide instructions, C/350 signed A-2
Dividend 3-8
Division by zero, attempting 2-8, 5-19f
Division instructions,
  fixed-point 2-5
  floating-point 3-8
Division,
  aborting floating-point 3-13
  floating-point 3-8
Divisor and dividend, comparing 3-8
Divisor equals zero 3-13
Divisor exponent 3-8
Divisor for zero, testing the 3-8
Divisor, defined 3-8
DIVS 2-5, 10-26
DIVX 2-5, 10-27
DLSH 2-14, 10-27
DMVA 2-21, 2-24, 5-22, 10-27, D-2
DMVC 2-21, 2-24, 5-22, 10-28
DMVF 2-21, 2-24, 5-22, 10-28, D-2
DMVN 2-21, 2-24, 5-22, 10-29, D-2
DMVO 2-21, 2-24, 5-22, 10-29, D-2
DMVS 2-21, 2-24, 5-22, 10-30, D-2
DNDF 2-21, 10-30
DO until greater than instructions 5-3
DO until greater than,
  narrow 10-79, 10-185f
  wide 10-87, 10-192
DOA 7-3, 10-30
DOB 7-3, 10-31
DOC 7-3, 10-31
DONE flag 7-3, 7-5
Double hex shift left, DHXL 2-22, 10-23
Double hex shift right, DHXR 2-22, 10-23f
Double logical shift 2-14, 10-27
Double precision 3-1f, 3-6f
Double to single, floating-point rounding 3-3, 10-53f
Double word addressed by WSP,
  decrement 2-9, 4-5, 10-34
  increment 2-9, 4-5, 10-65
  load accumulator with 4-5
  store accumulator in 2-3, 4-5
Double word block in a stack, accessing 4-4
Double word for vector stack, constructing 7-12
Double word in a stack, accessing 4-4
Double word onto vector stack, pushing 7-12
Double word operand 1-2, 1-13
Double word,
  load accumulator with 10-68
  pushing a 4-3
Double,
  add (FPAC to FPAC) 3-6, 10-42
  add (memory to FPAC) 3-6, 9-9, 10-42, 10-72, 10-179
  divide (FPAC by FPAC) 3-8, 10-44
  divide (FPAC by memory) 3-8, 9-9, 10-44, 10-72, 10-180
  load floating-point 9-9, 10-47, 10-73, 10-181
  multiply (FPAC by FPAC) 3-7, 10-49f
  multiply (FPAC by memory) 3-7, 9-9, 10-49, 10-74f, 10-181
  store floating-point 9-9, 10-59, 10-76, 10-183
  subtract (FPAC from FPAC) 3-7, 10-55
  subtract (memory from FPAC) 3-7, 9-9, 10-56, 10-75, 10-182
Double-word stack access instructions 4-5
Driver, device 7-2, 7-5
DSB 2-22, 10-31f
DSPA 9-10, 10-32f
DSSO 2-21, 10-33
DSSZ 2-21, 10-33
DSTK 2-21, 10-33
DSTO 2-21, 10-34
DSTZ 2-21, 10-34
DSZ 9-8, 10-34
DSZTS 2-9, 4-5, 10-34
DVZ flag (see INV flag) 3-13

E

ECLID 8-12, 10-35
ECLIPSE C/350 code 9-3f
EDIT 2-24, 5-21f, 9-8, 10-35f, D-2
Edit decimal and alphanumeric 16-bit data 9-8, 10-35f
EDIT instruction 2-21, 9-3
Edit subprogram 2-21, 5-4
Edit,
  end 10-22
  wide 4-8, 5-5, 9-8, 10-133ff
EDSZ 9-8, 10-36
Effective address 1-10f, 1-16, 2-20f, 5-2, 5-4f, 5-10, 7-6, 7-11, 8-2
Effective address,
  C/350 9-2, 9-5
  calculating an 2-21f
  extended load 9-10, 10-39
  load 2-12ff, 2-21f, 8-3, 9-10, 10-71, 10-77, 10-184
  push 2-21
Effective addressing 1-13
Effective addressing, C/350 9-5f
Effective byte address,
  load 1-13, 2-21, 10-77, 10-184
  push 2-20f, 4-5f, 4-8, 10-81, 10-189
Effective logical address 8-7
Effective relative address 1-11
Eight segment base registers, specifying 8-7
EISZ 9-8, 10-37
EJMP 9-10, 10-37

EJSR 9-10, 10-37
ELDA 9-8, 10-38
ELDB 9-8, 10-38
ELEF 9-10, 10-39
Element,
  data 6-1
  dequeue data 10-22
  dequeuing a data 6-5
  enqueuing a data 6-4
  index to a gate 5-11
  queue 5-4
Empty queue 6-3
Enable mask, trap 3-12f, 5-19
Enable,
  fixed-point trap 2-9, 10-61
  floating-point trap 3-12, 10-60
Enabling fixed-point fault recognition 2-10, 5-14
Enabling floating-point fault detection 3-13
Enabling floating-point fault recognition 3-11ff, 5-20
Enabling I/O instruction execution 7-1ff
Enabling I/O interrupt recognition 7-5, 7-12
Enabling narrow stack underflow 9-2
Enabling OVK mask 4-8
Enabling stack fault recognition 5-25
Enabling TE flag 4-8
Enabling vector stack underflow or overflow detection 7-9
Encountering an invalid digit 2-24, 5-22
End edit 2-21, 10-22
End float 2-21, 10-30
ENQH 6-5, 10-39f
ENQT 6-5, 10-40
Enqueue towards the head or tail 10-39f
Enqueuing data element 6-1, 6-4
Entry point to a segment 5-9f
Entry,
  addressing an 7-12
  page table 8-1ff
  vector table 7-12
Equivalent instruction 9-7ff
Error bits in the FPSR, setting 3-14
Error code in INP bits, setting 3-8
Error code, protection fault 8-8
Error codes, floating-point 3-10
Error flag,
  floating-point input argument 3-13
  setting an 3-6
Error,
  detecting an 5-13, 10-23
  floating-point arccosine instruction 3-14
  floating-point arcsine instruction 3-14
  floating-point divide by zero 3-14
  floating-point exponential instruction 3-14
  floating-point logarithm instruction 3-14
  floating-point power instruction 3-14
  floating-point square root instruction 3-14
  floating-point tangent instruction 3-14
  identifying floating-point 3-12, 3-14
Errors, clearing FPSR 3-12, 10-43
ESTA 9-8, 10-41
ESTB 9-8, 10-41
Example of one- or two-level page table translation 8-9f
Example,
  decimal arithmetic 2-25
  wide stack operation 5-7f
Exceeding processor storage capacity 1-2
Excess 64 notation, maintaining 3-7f
Excess 64 representation 3-2
Exchange accumulators 2-3, 10-178
Exchange, wide 2-3, 10-176
Exclusive OR 2-12f, 10-188
Exclusive OR immediate 2-13, 10-189
Exclusive OR immediate, wide 2-13, 10-177
Exclusive OR, wide 2-13, 10-177
Execute 10-179
Execute access 1-9
Execute access flag 8-6
Execute access violation 5-15
Execute accumulator instruction 5-2
Execute instruction, accessing page to 8-5
Execute protection violation 5-17
Execute violation fault code D-1
Execute,
  pop block and 2-9, 4-5, 4-8, 5-5f, 5-28, 9-10f, 10-101, 10-103
  wide pop block and 2-9, 4-5, 4-8, 5-8, 5-14, 5-26, 9-10f, 10-160f
Executing 16-bit instructions 9-1
Executing accumulator contents 5-2
Executing C/350 memory reference instruction 9-4ff
Executing floating-point instructions 3-4
Executing interrupted program 5-14
Executing I/O instruction 7-1ff
Executing jump instruction 5-1
Executing LEF instruction 7-1ff
Executing return instruction 5-1
Executing skip instruction 5-1
Executing subroutine 5-1, 5-12
Executing XCT instruction 5-1
Execution,
  aborting instruction 2-24, 8-2
  changing current segment of 7-6
  enabling I/O instruction 7-1ff
  normal program 5-19f
Exiting subroutine 4-3
Expanding beyond 64 Kbytes 9-3
Expanding data areas 9-3
Expanding ECLIPSE C/350 code 9-3f
Expanding the instruction set 4-5
Expanding to 32-bit fixed-point arithmetic 9-3f

Explanation of fault code D-1
Explicit data types 2-15ff, 2-24
Exponent 3-2
Exponent into FPAC, loading 3-3
Exponent overflow 3-6
Exponent overflow flag 3-12
Exponent underflow 3-6
Exponent underflow flag 3-12
Exponent,
  decrementing an intermediate 3-5
  dividend 3-8
  divisor 3-8
  intermediate floating-point 3-7
  load 10-45
  producing an intermediate 3-8
Exponential double, floating-point 10-140
Exponential instruction error, floating-point 3-14
Exponential single, floating-point 10-140
Exponential, floating-point 3-11
Exponentiation evaluation, floating-point 3-10f
Exponents 3-7
Extend 16-bits to 32-bits,
  sign 2-2, 10-109
  zero 2-2, 10-194
Extend and divide, sign 2-5, 10-27
Extend divide, narrow sign 2-5
Extend multiply, narrow sign 2-5
Extended add immediate 2-4, 10-5
Extended add immediate, narrow 2-4, 10-92
Extended decrement and skip if zero 9-8, 10-36
Extended displacement 1-11
Extended increment and skip if zero 9-8, 10-37
Extended jump 9-10, 10-37
Extended jump to subroutine 9-10, 10-37
Extended load accumulator 9-8, 10-38
Extended load byte 9-8, 10-38
Extended load effective address 9-10, 10-39
Extended operation 4-5, 4-8, 9-10f, 10-103, 10-188
Extended operation, wide 5-5, 9-10f, 10-176
Extended store accumulator 9-8, 10-41
Extended store byte 9-8, 10-41
Extended,
  load integer 9-9, 10-70
  store integer 9-9, 10-114
  wide load integer 9-9, 10-154
  wide store integer 9-9, 10-173
External device 7-4

F

FAB 3-3, 10-42
Facilities, maintaining I/O 7-5
FAD 3-6, 10-42
Failure, detecting power 7-5
FAMD 3-6, 9-9, 10-42
FAMS 3-6, 9-9, 10-42f
FAS 3-6, 10-43
Fault address parameter, vector stack 7-9
Fault code,
  access violation D-1
  ASCII or decimal 2-24, 5-14, 5-21f, D-2
  decimal or ASCII 2-24, 5-14, 5-21f, D-1f
  explanation of D-1
  invalid microinterrupt return block D-1
  inward return D-1
  loading AC1 with 5-24f, 9-3
  nonprivileged 5-14, D-1f
  outward call D-1
  privileged D-1
  privileged instruction violation D-1
  protection 8-8
  read violation D-1
  returned in AC1 2-23, 5-21, 5-23ff, 9-3, D-2
  unimplemented instruction D-1
  validity bit protection D-1
  wide stack 5-26, D-1f
  write violation D-1
Fault codes,
  decimal and ASCII 2-23
  protection 5-17
Fault detection, enabling floating-point 3-13
Fault flags,
  fixed-point 2-10
  floating-point 3-12, 5-20
Fault handler definition, protection 5-15f
Fault handler,
  C/350 decimal and ASCII 9-3
  C/350 floating-point 9-3
  decimal and ASCII 2-23, 5-21
  first instruction of 5-14, 5-20, 5-25, 5-28
  first instruction of interrupt 7-13
  first instruction of vector stack 7-13
  fixed-point 2-10
  floating-point 3-12, 5-21
  jumping to 5-13f, 5-19f
  jumping to narrow stack 5-28
  last instruction of 5-14, 5-26, 5-28
  narrow stack 5-27f
  page 5-13, 8-11f
  protection 5-15f
  returning from 3-13, 4-1
  saving pointer to wide stack 7-9
  vector stack 7-13
  wide stack 5-25f, 5-28
Fault instruction,
  address of 2-23, 5-14ff, 5-19ff, 5-25f, 5-28
  address of decimal 5-24
  address of WEDIT 5-23
  determining address of 5-21
Fault mask,
  setting fixed-point overflow 5-19
  setting floating-point 5-20
  TE 5-20

Fault operations, stack 5-25f
Fault pointer,
  initializing wide stack 7-9
  privileged and nonprivileged 5-13
Fault recognition,
  enabling fixed-point 2-10, 5-14
  enabling floating-point 3-11ff, 5-20
  enabling stack 5-25
Fault return block 5-15f
Fault return block,
  fixed-point 5-19
  floating-point 5-20
  pushing 5-14, 5-24
  stack 4-8, 5-26, 5-28
Fault return blocks,
  types of narrow 5-24
  types of wide 5-23
Fault service mask 1-2f
Fault subopcode, byte pointer to 5-24
Fault,
  aborting nonprivileged 5-14
  ASCII data 5-14, 5-21ff
  byte pointer to subopcode causing 5-23
  cause of page 8-11
  checking for stack overflow 4-3
  decimal data 2-24, 5-21ff
  decimal instruction address causing 5-23
  decimal or ASCII D-2
  detecting a 2-10, 3-13, 5-1
  detecting an overflow 2-10
  detecting floating-point 5-20
  detecting nonprivileged or privileged 5-13
  detecting wide stack 5-25
  disabling stack 4-7, 5-25, 5-27
  fixed-point overflow 1-2, 2-8, 5-13, 5-19
  floating-point 3-11, 5-13f
  ignoring floating-point overflow 5-20
  infinite protection 5-15
  initiating a protection 5-8
  initiating fixed-point overflow 5-19
  initiating floating-point 5-20
  instruction causing 2-24, 5-22
  microinterrupt or 5-26
  narrow stack 5-25f
  nonprivileged 1-16f, 5-13
  nonresident page 5-13
  page 8-5, 8-8, 8-11
  page table depth 8-7
  page table validity protection 8-8
  privileged 1-16f
  privileged access 5-12
  protection 1-8f, 2-22, 5-11, 5-13, 5-19, 8-7f
  segment validity protection 8-2
  servicing 1-2, 2-10
  servicing a floating-point 3-12f
  servicing floating-point 3-12f, 5-20
  servicing nonprivileged 5-14
  servicing overflow 2-10
  servicing page 8-5, 8-11
  servicing stack 5-25, 5-27
  stack 4-1, 5-12, 5-14, 5-16
  types of 1-16
  wide floating-point 5-20f
  wide stack 5-25
  words required beyond WSL for stack 4-8
Faulting instruction, first instruction after 5-19f
Faults and interrupts, C/350 9-3
Faults,
  C/350 commercial A-3
  data type 2-24
  floating-point 3-11ff, 5-20f
  handling 5-13
  priority of handling 5-13
  type of stack 5-25
  wide stack 4-7
FCLE 3-12, 3-14, 10-43
FCMP 3-8f, 10-43
FDD 3-8, 3-14, 10-44
FDMD 3-8, 3-14, 9-9, 10-44
FDMS 3-8, 3-14, 9-9, 10-44
FDS 3-8, 3-14, 10-45
FEXP 3-3, 10-45
FFAS 3-3, 3-13, 10-45
FFMD 3-13, 9-9, 10-46
FHLV 3-8, 10-46
Field in context block, status 8-11
Field,
  data type and size 2-16
  device address 7-4
  immediate 8-8
Final intermediate mantissa 3-7
Final interrupt processing 7-11
Finishing an I/O operation 7-5
FINT 3-3f, 10-46f
First 64 Kbytes of segment 9-2
First entry in page table, address of 8-2ff
First instruction after faulting instruction 5-19f
First instruction of fault handler 5-14, 5-20, 5-25, 5-28
First instruction of interrupt fault handler 7-13
First instruction of interrupt handler 7-7, 9-3
First instruction of subroutine, address 5-10, 5-12
First instruction of vector stack fault handler 7-13
First word for I/O transfer, address of 7-2
First word of interrupt handler 7-7
Fix from floating-point accumulator, wide 3-3f, 3-13, 9-9, 10-141
Fix to accumulator 10-45
Fix to memory 9-9, 10-46
Fixed-point accumulator 1-2

Fixed-point accumulator,
C/350 9-1
wide float from 3-3f, 9-9, 10-141
Fixed-point addition instructions 2-4, 2-22
Fixed-point arithmetic instructions 2-4
Fixed-point arithmetic, expanding to 32-bit 9-3f
Fixed-point byte movement instructions 2-20
Fixed-point computation 1-2, 1-16, 2-1
Fixed-point computation fault 5-13
Fixed-point conversion, floating-point to 2-20, A-1
Fixed-point data format 2-1
Fixed-point data movement instructions 2-3
Fixed-point data precision 2-2
Fixed-point decimal 3-3
Fixed-point decrement word and skip 2-9
Fixed-point division instructions 2-5
Fixed-point fault recognition, enabling 2-10, 5-14
Fixed-point fault return block 5-19
Fixed-point increment word and skip 2-9
Fixed-point instructions, C/350 9-7f
Fixed-point logical data format 2-12
Fixed-point logical instructions 2-12
Fixed-point logical operations 2-12
Fixed-point logical skip instructions 2-14
Fixed-point move instructions 2-3
Fixed-point multiplication instructions 2-5
Fixed-point overflow 2-6
Fixed-point overflow condition 2-10
Fixed-point overflow fault 1-2, 2-8, 5-19
Fixed-point overflow flag, setting 5-19
Fixed-point overflow mask, OVK 2-10, 9-10f
Fixed-point precision conversion 2-2
Fixed-point registers 1-2
Fixed-point skip on condition 2-7f
Fixed-point subtraction instructions 2-4
Fixed-point subtraction instructions, decimal 2-22
Fixed-point to floating-point conversion 3-3
Fixed-point trap disable 2-9, 10-61
Fixed-point trap enable 2-9, 10-61
Fixed-point two's complement 2-2
Flag (see INV flag), DVZ 3-13
Flag handling, optional device 7-3
Flag is zero, opcode pointer if sign 2-21
Flag of the FPSR,
INP 3-10
INV 3-10
N 3-10
OVF 3-10
UNF 3-10
Z 3-10
Flag reset, skip on OVR 2-8, 10-111
Flag to one or zero, set sign 2-21, 10-33
Flag,
@ 9-5f
ANY 3-12f, 5-20
BUSY 7-3, 7-5
CARRY 2-5f, 4-6, 5-6f, 5-19f, 5-23, 5-26, 9-2
changing a device 7-4
changing RND 3-4
controlling ION 7-6
device 7-3ff
DONE 7-3, 7-5
enabling TE 4-8
error status 3-12
execute access 8-6
exponent overflow or underflow 3-12
floating-point input argument error 3-13
indirect 1-15
initializing BUSY 7-3
initializing DONE 7-3
initializing OVR 5-19, 7-13
INP 3-8, 3-14
INP (invalid input indicator) 3-14
insert character depending on sign 2-21
instructions to manipulate modified or referenced 8-12
interrupt on 7-4ff
INV 3-8f, 3-13, 5-20
I/O validity 1-17, 7-2, 8-3
ION 7-4ff
IRES 2-10f, 5-19, 5-25, 7-6, 7-13, 9-3
IXCT 2-10f
LEF mode 8-3
manipulating a device 7-5
manipulating an interrupt on 7-5
manipulating modified 8-12
mantissa overflow 3-13
memory resident 8-5
modified 8-12
MOF 3-13, 5-20
N (negative flag) 3-8, 3-13
overflow 2-8, 2-10, 5-25, 7-13
OVF 3-6, 3-12, 5-20
OVR 2-8, 2-10, 5-19, 5-25, 7-13
power fail 7-4f
read access 8-6
referenced 8-12
RND or round 3-4f, 3-13
segment validity 8-2
setting an error 3-6
setting BUSY 7-5
setting DONE 7-5
setting fixed-point overflow 5-19
setting INV 3-8, 3-14
setting ION 7-6
setting modified 8-12
setting N 3-8
setting overflow 2-8
setting referenced 8-12
setting RND 3-13
setting Z (true zero flag) 3-8, 3-13
TE 4-8
TES 2-11

testing a device 7-3f
testing BUSY 7-3
testing DONE 7-3
testing INV 3-9
translation level 8-3
UNF 3-12, 5-20
valid page access 8-5
write access 8-6
Z (true zero flag) 3-13
Flags,
device 7-3ff
floating-point fault 3-12, 5-20
for general I/O devices 7-5
for skip instruction 7-5
setting floating-point fault 5-20
testing status 3-8
FLAS 3-3, 10-47
FLDD 3-4, 9-9, 10-47
FLDS 3-4, 9-9, 10-47
FLMD 9-9, 10-48
Float from accumulator 10-47
Float from accumulator, wide 3-3f, 9-9, 10-141
Float from memory 9-9, 10-48
Float,
end 2-21, 10-30
move 10-28
move j 2-21
Floating-point accumulator,
C/350 9-1
wide fix from 3-3f, 3-13, 9-9, 10-141
Floating-point addition instructions 3-6
Floating-point arccosine 3-11
Floating-point arccosine double 10-135
Floating-point arccosine instruction error 3-14
Floating-point arccosine single 10-135f
Floating-point arcsine 3-11
Floating-point arcsine double 10-136
Floating-point arcsine instruction error 3-14
Floating-point arcsine single 10-136f
Floating-point arctangent 3-11
Floating-point arctangent (two-accumulator) 3-11
Floating-point arctangent double 10-137
Floating-point arctangent double (two-accumulator) 10-138
Floating-point arctangent single 10-137
Floating-point arctangent single (two-accumulator) 10-138
Floating-point binary logarithm 3-11
Floating-point binary logarithm double 10-141f
Floating-point binary logarithm single 10-142
Floating-point carry 3-6
Floating-point common logarithm 3-11
Floating-point common logarithm double 10-143
Floating-point common logarithm single 10-144
Floating-point computation 1-3, 1-16, 2-15, 3-1
Floating-point computation fault 5-13
Floating-point conversion 3-4
Floating-point conversion, fixed-point to 2-20, 3-3f
Floating-point cosine 3-11
Floating-point cosine double 10-139
Floating-point cosine single 10-139
Floating-point data,
converting 2-20
loading 3-4
moving 3-4
normalizing 3-1
storing 3-4
Floating-point decimal conversion instructions 3-3
Floating-point divide by zero error 3-14
Floating-point divide instruction 3-13
Floating-point division 3-8
Floating-point division, aborting 3-13
Floating-point double,
load 9-9, 10-47, 10-73, 10-181
store 9-9, 10-59, 10-76, 10-183
Floating-point error codes 3-10
Floating-point error, identifying 3-12, 3-14
Floating-point exponent, intermediate 3-7
Floating-point exponential 3-11
Floating-point exponential double 10-140
Floating-point exponential instruction error 3-14
Floating-point exponential single 10-140
Floating-point exponentiation evaluation 3-10f
Floating-point fault 3-11, 5-21
Floating-point fault detection, enabling 3-13
Floating-point fault flags 3-12
Floating-point fault flags, setting 5-20
Floating-point fault handler 3-12, 5-21
Floating-point fault handler, C/350 9-3
Floating-point fault mask, setting 5-20
Floating-point fault recognition, enabling 3-11ff, 5-20
Floating-point fault return block 5-20
Floating-point fault,
initiating 5-20
servicing 3-12f, 5-20
Floating-point faults 5-20f
Floating-point format 3-2
Floating-point halve (FPAC/2) 3-8, 10-46
Floating-point identification code 3-13
Floating-point illegal input argument 3-10
Floating-point input argument error flag 3-13
Floating-point instruction, address of 3-14
Floating-point instructions,
C/350 9-9
executing 3-4
Floating-point intrinsic instruction format 3-10
Floating-point invalid input fault, detecting 5-20
Floating-point inverse trigonometric functions 3-10f
Floating-point logarithm instruction error 3-14
Floating-point move instructions 3-3
Floating-point multiplication 3-7

Floating-point multiply double or single precision 3-7
Floating-point natural logarithm 3-11
Floating-point natural logarithm double 10-142
Floating-point natural logarithm single 10-142
Floating-point normalize 3-3
Floating-point number, scaling 3-3, 10-54
Floating-point numbers, compare two (set N and Z) 3-9
Floating-point numerical algorithms A-2
Floating-point pop, wide 3-4, 4-5, 4-8, 10-144f
Floating-point power 3-11
Floating-point power double 10-147
Floating-point power instruction error 3-14
Floating-point power single 10-147
Floating-point program counter 3-8, 3-10, 3-14
Floating-point push, wide 3-4, 4-5, 4-8, 10-146
Floating-point result, storing the 3-6
Floating-point revision 3-13
Floating-point rounding double to single 3-3, 10-53f
Floating-point sine 3-11
Floating-point sine double 10-148
Floating-point sine single 10-148
Floating-point single,
  load 9-9, 10-47, 10-73, 10-181
  store 9-9, 10-60, 10-76, 10-183
Floating-point skip on condition instructions 3-9
Floating-point square root 3-11
Floating-point square root double 10-149
Floating-point square root evaluation 3-10f
Floating-point square root instruction error 3-14
Floating-point square root single 10-149
Floating-point state instruction, load 3-14
Floating-point state,
  pop 3-12, 9-9, 10-51f
  push 3-12, 9-9, 10-52f
Floating-point status 3-11ff
Floating-point status and accumulators,
  pop 10-51f
  push 10-52f
Floating-point status instructions 3-12
Floating-point status register 1-3, 3-4, 3-6, 3-12f, 9-9, 10-59, 10-75
Floating-point status,
  load 10-74
  store 5-21
Floating-point subtraction 3-7
Floating-point tangent 3-11
Floating-point tangent double 10-151
Floating-point tangent instruction error 3-14
Floating-point tangent single 10-151
Floating-point to decimal, converting 3-3
Floating-point to fixed-point conversion 2-20, A-1
Floating-point to fixed-point, conversion 3-3
Floating-point trap 3-12, 10-60, A-2
Floating-point trigonometric functions 3-10f
Floating-point word, reading high 3-3
Floating-point,
  conversion fixed-point to 3-3
  integerize 3-3, 10-46f
  move 10-50
  scale 3-3
FLST 9-9, 10-48
FMD 3-7, 10-49f
FMMD 3-7, 9-9, 10-49
FMMS 3-7, 9-9, 10-49
FMOV 3-4, 10-50
FMS 3-7, 10-50
FNEG 3-3, 10-50
FNOM 3-1, 3-3, 10-50
FNS 5-3, 10-51
Format,
  ASCII and decimal data 2-15ff
  byte pointer 1-14
  data 2-15
  double precision 3-2
  fixed-point data 2-1
  fixed-point logical 2-12
  floating-point 3-2
  map register assignments and 7-2
  normalized E-3
  processor status register 2-10
  program counter 1-5
  single precision 3-2
Formatted data, transferring 2-20
Forming bit pointer 9-6
Forming byte address 9-5f
Forming physical address 7-2
Forward link 6-2
Forward search queue and skip,
  narrow 5-3, 6-5, 10-96f
  wide 5-3f, 6-5, 10-150
Four FPAC's and load in memory, convert the 2-20
Four wide stack registers, initializing 7-9, 7-11
FPAC data and load in memory, convert 2-20
FPAC,
  converting a decimal and loading in 2-20
  loading exponent into 3-3
FPOP 9-9, 10-51f
FPPC (see floating-point program counter) 3-8, 3-14
FPSH 9-9, 10-52f
FPSR 3-11ff
FPSR errors, clearing 3-12, 10-43
FPSR, setting error bits in the 3-14
Frame (frame number), root page table 8-3
Frame pointer,
  initializing wide 4-3, 5-11
  loading wide 5-12f
  narrow 9-2
  saving wide 7-9
  storing wide 5-7, 5-11, 5-13, 8-11
  wide 4-2f, 5-6ff, 5-15, 7-9

Frame, root page table 8-2
FRDS 3-3, 10-53f
Frequency of references to pages 8-12
FRH 3-3, 10-54
FSA 5-3, 10-54
FSCAL 3-3f, 3-13, 10-54
FSD 3-7, 10-55
FSEQ 3-8f, 10-55
FSGE 3-9, 10-55
FSGT 3-8f, 10-55
FSLE 3-9, 10-55
FSLT 3-8f, 10-56
FSMD 3-7, 9-9, 10-56
FSMS 3-7, 9-9, 10-56
FSND 3-9, 10-57
FSNE 3-9, 10-57
FSNER 3-9, 10-57
FSNM 3-9, 10-57
FSNO 3-9, 10-57
FSNOD 3-9, 10-58
FSNU 3-9, 10-58
FSNUD 3-9, 10-58
FSNUO 3-9, 10-58
FSS 3-7, 10-59
FSST 5-21, 9-9, 10-59
FSTD 3-4, 9-9, 10-59
FSTS 3-4, 9-9, 10-60
FTD 3-12, 5-20, 10-60
FTE 3-12, 5-20, 10-60
FTE mask 3-13
Functional capabilities 1-1
FXTD 2-9, 5-19, 10-61
FXTE 2-9f, 5-19, 10-61

G

Gate array 5-10f
Gate element, index to a 5-11
Gate fault code, illegal D-1
Gate number 5-10f
Gate protection violation, illegal 5-17
Gate,
  indexed 5-11
  valid 5-11
General I/O devices, device flags for 7-5
General I/O instructions 7-3f
Glossary E-1
Guard digit 3-4f

H

Halve,
  floating-point (FPAC/2) 3-8, 10-46
  narrow (AC/2) 2-5, 10-61
  wide (AC/2) 2-5, 10-152
Handler,
  breakpoint 2-9, 4-5, 4-8, 5-5f
  C/350 decimal and ASCII fault 9-3
  C/350 floating-point fault 9-3
  fault 2-10, 2-23, 5-21, 8-8
  first instruction of fault 5-14, 5-20, 5-25, 5-28
  first instruction of interrupt 7-7, 9-3
  first instruction of interrupt fault 7-13
  first instruction of vector stack fault 7-13
  first word of interrupt 7-7
  floating-point fault 3-12, 5-21
  immediate interrupt 7-7
  interrupt 7-5f, 10-190
  jumping to fault 5-13f, 5-19f
  jumping to narrow stack fault 5-28
  last instruction of fault 5-14, 5-26, 5-28
  last instruction of vectored interrupt 7-7
  narrow stack fault 5-27f
  page fault 5-13, 8-11f
  returning from breakpoint 4-5
  returning from fault 3-13, 4-1
  saving pointer to wide stack fault 7-9
  subroutine 2-10
  vector stack fault 7-13
  vectored interrupt 7-7, 10-190
  wide stack fault 5-25f
Handling faults, priority of 5-13
Handling I/O interrupts, priority of 5-14
Head, enqueue towards the 10-39f
Hex digit 3-4
Hex shift left,
  DHXL double 2-22, 10-23
  HXL single 2-22
  single 10-61
Hex shift right,
  DHXR double 2-22, 10-23f
  HXR single 2-22
  single 10-62
Hierarchical interrupt system 7-5
Hierarchical protection mechanism 1-17
High floating-point word, reading 3-3
High speed device and memory, transfers between 7-1
High word, read 10-54
HLV 2-5, 10-61
Honoring interrupts 7-13
HXL 10-61
HXL single hex shift left 2-22
HXR 10-62
HXR single hex shift right 2-22

I

I/O instruction violation 5-15
I/O interrupt during protection violation 5-17
I/O operation, invalid 5-14ff
I/O protection violation 5-17
ID code 3-13

Identical segments, testing for 5-12
Identification code, floating-point 3-13
Identification instructions 8-12
Identification,
  C/350 load CPU 8-12, 10-35
  long load CPU 8-12, 10-67
  narrow load CPU 8-12, 10-94
Identifier field, bit 1-15
Identifier, bit 1-15, 9-6f
Identifying floating-point error 3-12, 3-14
Identifying unique device 7-4
Idle device 7-5
Ignoring an interrupt request 7-5
Ignoring overflow faults 5-19
Ignoring page protocol access 8-7f
IIS (see intrinsic instruction set) 3-10f
Illegal gate fault code D-1
Illegal gate protection violation 5-17
Illegal input argument, floating-point 3-10
Illegal inward return 5-9
Illegal I/O operation 8-3
Illegal outward subroutine call 5-9
Immediate field 8-8
Immediate interrupt handler 7-7
Immediate,
  add 2-4, 10-5f
  AND 2-13, 10-8
  command I/O 7-3, 10-11
  exclusive OR 2-13, 10-189
  extended add 2-4, 10-5
  inclusive OR 2-13, 10-64
  insert characters 2-21, 10-25
  narrow add 2-4, 10-78, 10-93, 10-185
  narrow extended add 2-4, 10-92
  narrow load 2-3, 10-98
  narrow subtract 10-80, 10-100, 10-187
  subtract 2-4, 10-108
  wide add 2-4, 10-86, 10-119, 10-191
  wide add with narrow 10-159
  wide add with wide 10-119
  wide AND 2-13, 10-120
  wide arithmetic shift with narrow 10-121
  wide exclusive OR 2-13, 10-177
  wide inclusive OR 2-12f, 10-152
  wide load with wide 10-153
  wide logical shift 2-14, 10-156
  wide logical shift left 2-14
  wide logical shift with narrow 10-156
  wide skip if accumulator equal to 2-23
  wide skip if accumulator greater than 2-8, 10-167
  wide skip if accumulator less than or equal to 2-8, 10-169
  wide skip if accumulator not equal to 2-8, 10-170
  wide skip if equal to 2-8
  wide subtract 10-88, 10-166, 10-193
  wide unsigned skip if accumulator greater than 2-8, 10-174
  wide unsigned skip if accumulator less than or equal to 2-8, 10-175
Implicit data structure 2-15
Impure zero E-2
INC 2-4, 2-6, 2-8f, 10-62f
Inclusive OR 10-64
Inclusive OR immediate 2-13, 10-64
Inclusive OR immediate, wide 2-12f, 10-152
Inclusive OR, wide 2-13, 10-152
Increasing accuracy of result 3-4
Increment and skip 10-62f
Increment and skip if zero,
  C/350 9-8, 10-64
  extended 9-8, 10-37
  narrow 2-9, 9-8, 10-79, 10-186
  wide 2-9, 10-88, 10-192
Increment double word addressed by WSP 2-9, 4-5, 10-65
Increment with optional skip 2-4, 2-6, 2-8f
Increment word and skip, fixed-point 2-9
Increment,
  wide (no skip) 2-4, 10-152
  wide decimal 2-22, 10-131
Incrementing a DO-loop variable 5-3
Incrementing interrupt level word 7-9
Incrementing program counter 3-9, 5-1
Incrementing wide stack pointer 4-3f
Independent operations, device 7-3
Index field 1-10
Index to a gate element 5-11
Indexed gate 5-11
Indicator for STIX, destination 5-23
Indicator for WSTI, destination 5-23
Indicator,
  byte 1-13, 9-5f
  destination 2-21
  explicit data type 2-16
  original source 5-23f
  source 2-21
Indirect 8-2
Indirect addressing 1-8, 1-10f, 1-15, 8-7, 9-5f
Indirect addressing violation 5-15
Indirect bit 9-5f
Indirect field 1-11
Indirect flag 1-15
Indirect logical address 8-7
Indirect pointer 1-11, 2-23, 5-10, 5-13, 5-19ff, 5-25, 5-28, 9-7, 9-9ff
Indirect relative address 1-11
Indirect, jumping 3-12, 7-7, 8-11
Indirection, levels of 5-13
Infinite loop during protection violation 5-16
Infinite protection fault 5-15

Information, interrupt routine dependent 7-12
Initializing burst multiplexor channel transfer 7-2
Initializing BUSY flag 7-3
Initializing carry flag instructions 2-6
Initializing data channel transfer 7-2
Initializing device map 7-2
Initializing DONE flag 7-3
Initializing four wide stack registers 7-9, 7-11
Initializing IRES flag 7-13
Initializing narrow stack 5-27
Initializing OVK mask 5-19, 7-13
Initializing OVR flag 5-19, 7-13
Initializing wide frame pointer 4-3, 5-11
Initializing wide stack 4-3
Initializing wide stack base 4-2, 8-11
Initializing wide stack fault pointer 7-9
Initializing wide stack limit 8-11
Initializing wide stack pointer 4-3, 8-11
Initializing wide stack registers 4-2, 4-4, 4-6
Initiating a protection fault 5-8
Initiating a transfer to another segment 5-8
Initiating fixed-point overflow fault 5-19
Initiating floating-point fault 5-20
Initiating I/O operation 7-5
INP bits 5-20
INP (invalid input indicator) flag 3-14
INP flag 3-8, 3-14
INP flag of the FPSR 3-10
Input argument (INV = 0), skip on no invalid 10-57
Input argument error flag, floating-point 3-13
Input argument, floating-point illegal 3-10
Input (OVF and INV = 0) argument, skip on no overflow and no invalid 10-58
Input (UNF and INV = 0) argument, skip on no underflow and no invalid 10-58
Input fault, detecting floating-point invalid 5-20
Insert character depending on sign flag 2-21
Insert character depending on trigger 2-21
Insert character j times 2-21, 10-25
Insert character once 2-21, 10-25
Insert character suppress 10-25
Insert characters immediate 2-21, 10-25
Insert sign 10-25
Inserting additional bytes 2-21
Inserting bytes 2-15
Inserting string of characters 2-20
Instruction address causing fault, decimal 5-23
Instruction after faulting instruction, first 5-19f
Instruction causing fault 2-24, 5-22
Instruction dictionary 10-1
Instruction error,
  floating-point arccosine 3-14
  floating-point arcsine 3-14
  floating-point exponential 3-14
  floating-point logarithm 3-14
  floating-point power 3-14
  floating-point square root 3-14
  floating-point tangent 3-14
Instruction execution,
  aborting 2-24, 8-2
  enabling I/O 7-1ff
Instruction fault code, unimplemented D-1
Instruction of fault handler,
  first 5-14, 5-20, 5-25, 5-28
  last 5-14, 5-26, 5-28
Instruction of interrupt fault handler, first 7-13
Instruction of interrupt handler,
  first 7-7, 9-3
  last 7-7
Instruction of subroutine,
  address first 5-10, 5-12
  last 5-13
Instruction of vector stack fault handler, first 7-13
Instruction of vectored interrupt handler, last 7-7
Instruction opcode, valid 5-19
Instruction opcodes,
  I/O 7-4
  wide A-1
Instruction operation, I/O 7-4
Instruction prefix, X or L 1-11
Instruction protection violation,
  privileged 5-17
  unimplemented 5-17
Instruction results, C/350 A-1
Instruction set E-2
Instruction set, expanding the 4-5
Instruction violation fault code, privileged D-1
Instruction violation,
  I/O 5-15
  privileged 5-15
  unimplemented 5-15
Instruction,
  aborting memory reference 8-2
  accessing page to execute 8-5
  accumulator jump 5-2
  accumulator skip 5-2
  address of decimal fault 5-24
  address of fault 2-23, 5-19ff, 5-26, 5-28
  address of floating-point 3-14
  address of WEDIT fault 5-23
  addressing next 5-1
  C/350 equivalent 9-8, 9-11
  C/350 memory reference 9-5f
  character move 5-12
  CPU device 7-6
  determining address of fault 5-21
  device flags for skip 7-5
  DO-loop 5-3
  EDIT 9-3
  equivalent 9-7, 9-9f

equivalent 32-bit processor 9-7
execute accumulator 5-2
executing C/350 memory reference 9-4ff
executing I/O 7-1ff
executing jump 5-1
executing LEF 7-1ff
executing return 5-1
executing skip 5-1
executing XCT 5-1
first instruction after faulting 5-19f
floating-point divide 3-13
format general I/O 7-4
INTDS CPU device 7-6
INTEN CPU device 7-6
interrupting an 7-6
I/O 7-2ff, 8-3
issuing an I/O 7-5
load effective address 8-3
load floating-point state 3-14
load physical address 7-2
mask out 7-6, 7-12
memory reference 1-8
port select 7-2
restarting interrupted 7-6
resume memory reference 8-5
resuming interrupted 7-6
size 1-8
time of fault address of fault 5-14ff
vector interrupt 7-7
wide return 5-6f, 5-13, 9-3f
wide save 4-3, 5-12
wide special save 5-5ff, 9-3f
Instructions for BMC maps, I/O 7-3
Instructions for data channel maps, I/O 7-3
Instructions to manipulate modified flag 8-12
Instructions to manipulate referenced flag 8-12
Instructions,
byte compare 2-15
byte move 2-20f
C/350 9-4
C/350 fixed-point 9-7f
C/350 floating-point 9-9
C/350 memory reference 9-1, 9-4ff
C/350 program flow 9-2
C/350 signed divided A-2
C/350 stack 9-11
C/350 stack management 9-11
C/350 stack-referenced 9-1
central processor identification 8-12
convert decimal/floating-point 2-20
decimal arithmetic 2-22
decimal fixed-point subtract 2-22
decimal move 2-20f
decimal move and convert 2-20
decimal shift 2-22
decimal string arithmetic 2-22
DO until greater than 5-3
double-word stack access 4-5
EDIT subprogram 2-21
executing 16-bit 9-1
executing floating-point 3-4
fixed-point add 2-22
fixed-point addition 2-4
fixed-point arithmetic 2-4
fixed-point byte movement 2-20
fixed-point data movement 2-3
fixed-point division 2-5
fixed-point logical 2-12
fixed-point logical skip 2-14
fixed-point move 2-3
fixed-point multiplication 2-5
fixed-point subtraction 2-4
floating-point addition 3-6
floating-point arithmetic 3-6
floating-point binary conversion 3-3
floating-point data move 3-4
floating-point decimal conversion 3-3
floating-point division 3-8
floating-point move 3-3
floating-point skip on condition 3-9
floating-point status 3-12
general I/O 7-3
hex shift 2-22
initializing carry flag 2-6
interrupting specific 2-11
jump 5-1f
load effective byte address 2-21
load effective word address 2-21
logical 2-13
logical shift 2-14
logical skip on condition 2-14
move decimal/floating-point 2-20
multi-word wide stack 4-8
NIO A-2
noninterruptible 7-6
privileged 8-2, 8-12
PSR manipulation 2-9
queue 6-5
search queue 5-4
segment transfer 5-8
sequence of subroutine 5-6
shift 2-6
skip 2-7, 2-14, 2-22f, 3-8, 5-2f
subroutine 5-5
system identification 8-12
unimplemented 5-19
wide E-2
wide arithmetic shift 2-6
wide skip on accumulator bit 2-23
wide stack 4-4f
wide stack return block 4-5

INTDS CPU device instruction 7-6
Integer extended,
load 9-9, 10-70
store 9-9, 10-114
wide load 9-9, 10-154
wide store 9-9, 10-173
Integer,
convert to 16-bit 10-19
converting a decimal 2-21
load 9-9, 10-69
store 9-9, 10-113f
wide load 9-9, 10-153
wide store 10-172
Integerize floating-point 3-3, 10-46f
INTEN CPU device instruction 7-6
Intermediate exponent,
decrementing an 3-5
producing an 3-8
Intermediate floating-point exponent 3-7
Intermediate level interrupt processing 5-8, 7-9ff, 10-160f
Intermediate logical address 1-11
Intermediate mantissa 3-5ff
Intermediate result,
rounding the 3-13
shifting an 2-6
signs of the 3-5
truncating 3-13
Internal device 7-4
Internal processor state 7-9
Interpreting LEF and I/O opcodes 7-1ff
Interrupt communication 7-5
Interrupt during protection violation, I/O 5-17
Interrupt fault handler, first instruction of 7-13
Interrupt handler 7-5f, 9-3, 10-190
Interrupt level word 7-9
Interrupt mask 7-5f, 7-12f
Interrupt on flag 7-4ff
Interrupt processing 5-6, 5-8, 7-7, 7-9, 7-11
Interrupt recognition 7-5, 7-12
Interrupt request,
blocking an 7-5f
detecting an I/O 5-1
I/O 2-10, 4-1
servicing an 7-5
Interrupt resume flag (see IRES flag)
Interrupt routine 7-12f
Interrupt routine dependent information 7-12
Interrupt sequence 7-8
Interrupt service routine 7-5
Interrupt service, concluding vector 7-11
Interrupt system 1-5, 7-5, 8-11
Interrupt,
intermediate level 5-8
I/O 5-6, 5-8, 5-14, 5-25, 5-28
I/O vector 2-9
returning from I/O 4-1, 5-6, 5-8
servicing an 7-6
servicing base level vector 7-9, 10-161
servicing intermediate level vector 7-11ff, 10-160f
servicing vector 7-3
vector on I/O 4-8
wide restore from an I/O 4-5, 5-8, 7-7
Interrupt-executed opcode flag 2-10f
Interrupted instruction 7-6
Interrupted program,
executing 5-14
restarting 8-11
Interrupting an instruction 7-6
Interrupting device number 7-12f
Interrupting device, vector on 10-190
Interrupting specific instructions 2-11
Interrupts 7-5
Interrupts,
C/350 faults and 9-3
honoring 7-13
I/O 9-3
priority of handling I/O 5-14
Intrinsic instruction format, floating-point 3-10
Intrinsic instruction set 3-10f
INV flag 3-8f, 3-13, 5-20
INV flag of the FPSR 3-10
INV flag, setting 3-14
Invalid ASCII data format 5-13
Invalid data type 6 or 7 2-24, 5-22, D-2
Invalid decimal data 1-16
Invalid decimal data format 5-13
Invalid digit 2-24, 5-22, D-2
Invalid I/O operation 5-14ff
Invalid input argument (INV = 0), skip on no 10-57
Invalid input (OVF and INV = 0) argument, skip on no overflow and no 10-58
Invalid input (UNF and INV = 0) argument, skip on no underflow and no 10-58
Invalid input fault, detecting floating-point 5-20
Invalid memory reference 5-14ff
Invalid microinterrupt return block fault code D-1
Invalid microinterrupt return block protection violation 5-17
Invalid opcode 2-24, 5-22, D-2
Invalid page 8-5
Invalid segment 8-2
Invalid sign code 2-24, 5-22, D-2
Inverse trigonometric functions, floating-point 3-10f
Invoking interrupt system 7-5, 8-11
Inward address reference fault code D-1
Inward address reference protection violation 5-17
Inward call 5-8

Inward reference violation 5-15
Inward return fault code D-1
Inward return protection violation 5-17
Inward return, illegal 5-9
I/O access, validate 8-2
I/O channel mask 7-12
I/O command, issuing programmed 7-3
I/O controller, communicating with 7-3
I/O data transfer 7-1ff
I/O immediate, command 7-3, 10-11
I/O instruction 7-2ff, 8-3
I/O instruction execution, enabling 7-1ff
I/O instructions, general 7-3
I/O interrupt 5-6, 5-8, 5-14, 5-25, 5-28, 9-3
I/O interrupt recognition 7-12
I/O interrupt request 2-10, 4-1
I/O interrupt request, detecting an 5-1
I/O interrupt,
  returning from 5-6, 5-8
  vector on 4-8
  wide restore from an 5-8
I/O interrupts, priority of handling 5-14
I/O memory reference 8-12
I/O mode, defined 7-1ff
I/O opcodes, interpreting LEF and 7-1ff
I/O operation,
  initiating and finishing an 7-5
  legal 8-3
I/O port (see I/O channel)
I/O protection violation fault code D-1
I/O reset 2-10, 7-3
I/O skip 7-3, 10-110
I/O system calls 7-5
I/O transfer 7-2f, 10-97
I/O validity flag 1-17, 7-2, 8-3
I/O vector interrupt 2-9
I/O,
  burst multiplexor channel 1-5f, 7-1f
  command 7-3, 10-11
  data channel 1-5f, 7-1ff
  programmed 1-5, 7-1, 10-102
ION flag 7-4ff
IOR 2-12f, 10-64
IORI 2-13, 10-64
IORST 3-14, 7-3
IRES flag,
  defined 2-10f
  initializing 7-13
  saving state of 5-19
  setting 5-25, 7-6, 9-3
Issuing an I/O instruction 7-5
Issuing programmed I/O command 7-3
Issuing read or write command to device map 7-3
ISZ 9-8, 10-64
ISZTS 2-9, 4-5, 10-65
IXCT flag, defined 2-10f

J

J alphabetical characters, move 2-21
J characters, move 2-21
J float, move 2-21
J numerics, move 2-21
J times, insert character 2-21, 10-25
JMP 9-10, 10-65
JSR 9-4, 9-10f, 10-65
Jump 9-4, 9-10, 10-65, 10-76, 10-183
Jump (with extended displacement) 5-2
Jump (with long displacement) 5-2
Jump if nonzero, decrement and 10-21
Jump indirect 3-12, 7-7, 8-11
Jump instructions 5-1f
Jump relative to program counter 9-10
Jump to fault handler 5-13f, 5-19f
Jump to stack fault handler 5-28
Jump to subroutine 5-4ff, 9-3f, 9-10, 10-65, 10-76, 10-183
Jump to subroutine,
  extended 9-10, 10-37
  push and 4-5, 4-8, 5-6
Jump,
  extended 9-10, 10-37
  pop PC and 9-10f, 10-103
  push 4-8, 5-5, 9-10f, 10-82, 10-104, 10-189
  wide pop PC and 4-5, 4-8, 5-5, 9-10f, 10-161

K

Kbyte address boundary, two 8-3
Kernel operating system 1-6

L

L instruction prefix 1-11
Large array, expanding data areas for 9-3
Last instruction of fault handler 5-14, 5-26, 5-28
Last instruction of interrupt handler 7-7
Last instruction of subroutine 5-13
LCALL 2-9, 4-4ff, 4-8, 5-4ff, 10-66f
LCALL count 4-6
LCPID 8-12, 10-67
LDA 9-8, 10-67
LDAFP 4-4, 10-67
LDASB 4-4, 10-68
LDASL 4-4, 10-68
LDASP 4-4, 10-68
LDATS 2-3, 4-5, 10-68
LDB 9-8, 10-69
LDI 9-9, 10-69
LDIX 2-24, 5-22, 9-9, 10-70, D-2
LDSP 5-2, 9-10, 10-70f
Lead bit,
  locate 2-13, 10-81
  locate and reset 2-13, 10-83

wide locate 2-13, 10-155
wide locate and reset 2-13, 10-155
Least significant bit 1-2, 2-6f
Least significant digit 2-17
LEF 8-3, 9-10, 10-71
LEF and I/O opcodes, interpreting 7-1ff
LEF instruction, executing 7-1ff
LEF mode flag 8-3
LEF mode,
defined 7-1ff, 8-3
selecting 7-1ff
Left immediate, wide logical shift 2-14
Left,
DHXL double hex shift 2-22, 10-23
HXL single hex shift 2-22
shifting one bit to the 2-6
single hex shift 10-61
Legal I/O operation 8-3
Level one, page 8-8
Level two, page 8-8
Level,
microcode revision 8-12
translation 8-3
Levels of indirection 5-13
LFAMD 3-6, 10-72
LFAMS 3-6, 10-72
LFDMD 3-8, 3-14, 10-72
LFDMS 3-8, 3-14, 10-73
LFLDD 3-4, 10-73
LFLDS 3-4, 10-73
LFLST 3-4, 3-12f, 9-9, 10-74
LFMMD 3-7, 10-74
LFMMS 3-7, 10-74f
LFSMD 3-7, 10-75
LFSMS 3-7, 10-75
LFSST 3-12, 5-21, 9-9, 10-75
LFSTD 3-4, 10-76
LFSTS 3-4, 10-76
Limit,
bit zero of narrow stack 5-27
bit zero of wide stack 5-26
initializing wide stack 8-11
loading wide stack 5-11, 5-13
lower stack 4-2
narrow stack 5-26ff, 9-2
setting bit zero of narrow stack 5-27f
upper stack 4-2
wide stack 4-3, 4-7, 5-11, 5-13, 5-25, 7-9, 7-11, 8-11
zero-extending vector stack 7-9
Limits and skip,
compare to 9-8, 10-12
wide compare to 2-8, 9-8, 10-124
Limits of narrow stack 9-2
Limits of wide stack 4-2f
Link 6-2
LJMP 5-2, 7-7, 10-76
LJSR 5-4ff, 9-3f, 10-76
LLDB 2-20, 10-77
LLEF 2-12ff, 2-21, 10-77
LLEFB 2-21, 10-77
LMRF 8-12, 10-77
LNADD 2-4, 10-78
LNADI 2-4, 10-78
LNDIV 2-5, 10-78
LNDO 5-3, 10-79
LNDSZ 2-9, 10-79
LNISZ 2-9, 10-79
LNLDA 2-3, 10-80
LNMUL 2-5, 10-80
LNSBI 2-4, 10-80
LNSTA 2-3, 10-80
LNSUB 2-4, 10-81
Load accumulator 9-8, 10-67
Load accumulator with double word 10-68
Load accumulator with double word addressed by WSP 4-5
Load accumulator with WFP 4-4, 10-67
Load accumulator with WSB 4-4, 10-68
Load accumulator with WSL 4-4, 10-68
Load accumulator with WSP 4-4, 10-68
Load accumulator,
double word addressed by WSP 2-3
extended 9-8, 10-38
narrow 2-3, 9-8, 10-80, 10-186
wide 2-3, 10-88, 10-193
Load all segment base registers 10-84
Load byte 2-20, 9-8, 10-69, 10-77, 10-184
Load byte,
extended 9-8, 10-38
wide 2-20, 9-8, 10-153
Load CPU identification,
C/350 8-12, 10-35
long 8-12, 10-67
narrow 8-12, 10-94
Load effective address 2-12ff, 2-21f, 8-3, 9-10, 10-71, 10-77, 10-184
Load effective address, extended 9-10, 10-39
Load effective byte address 1-13, 2-21, 10-77, 10-184
Load effective byte address instructions 2-21
Load effective word address instructions 2-21
Load exponent 10-45
Load floating-point double 9-9, 10-47, 10-73, 10-181
Load floating-point single 9-9, 10-47, 10-73, 10-181
Load floating-point state instruction 3-14
Load floating-point status 9-9, 10-48, 10-74
Load immediate, narrow 2-3, 10-98
Load in memory 2-20
Load integer 9-9, 10-69
Load integer extended 9-9, 10-70
Load integer extended, wide 9-9, 10-154

Load integer, wide 9-9, 10-153
Load map, wide 10-154
Load modified and referenced bits 8-12, 10-77
Load page table entry 8-5, 10-83
Load physical address instruction 7-2
Load physical and conditional skip 10-82
Load processor status register in AC0 10-83
Load PSR in AC0 2-9f
Load sign 9-8, 10-85f
Load sign, wide 2-13, 2-23, 9-8, 10-156f
Load with wide immediate, wide 10-153
Loading AC0 with address of fault 5-25
Loading AC1 with fault code 5-21, 5-23ff, 9-3
Loading AC2 with base address of DCT 7-12
Loading byte in accumulator 2-20
Loading byte in memory 2-20
Loading device map 7-3
Loading exponent into FPAC 3-3
Loading fault code in AC1 2-23, 5-21, 5-23ff, 9-3
Loading floating-point data 3-4
Loading FPSR 3-12
Loading in FPAC, converting a decimal and 2-20
Loading physical address into accumulator 7-3
Loading program counter 5-13
Loading referenced page from disk 8-11
Loading segment base register 8-2
Loading the vector stack 7-9
Loading wide frame pointer 5-12f
Loading wide stack base 5-11, 5-13
Loading wide stack limit 5-11, 5-13
Loading wide stack pointer 5-11, 5-13
LOB 2-13, 10-81
Locate and reset lead bit 2-13, 10-83
Locate and reset lead bit, wide 2-13, 10-155
Locate lead bit 2-13, 10-81
Locate lead bit, wide 2-13, 10-155
Location of narrow stack 5-27
Location of wide stack, top 4-3
Location,
  binary point 3-2
  defining stack 4-2
  updating WSP reserved memory 5-25
Logarithm double,
  floating-point binary 10-141f
  floating-point common 10-143
  floating-point natural 10-142
Logarithm instruction error, floating-point 3-14
Logarithm single,
  floating-point binary 10-142
  floating-point common 10-144
  floating-point natural 10-142
Logarithm,
  floating-point binary 3-11
  floating-point common 3-11
  floating-point natural 3-11
Logic instructions 2-13
Logical address 5-1, 8-2f, 8-5, 8-7, E-1
Logical address space 8-1, 8-11
Logical address,
  intermediate 1-11
  most significant bits translating 7-2
  translating 1-7, 7-2f, 8-2, 8-5
Logical data formats 2-12
Logical instructions 2-12f
Logical memory 8-1
Logical negate, optional shift 2-14
Logical one's complement, optional shift 2-14
Logical operations 2-1, 2-12
Logical shift 2-14, 10-85
Logical shift immediate, wide 2-14, 10-156
Logical shift instructions 2-14
Logical shift left immediate, wide 2-14
Logical shift with narrow immediate, wide 10-156
Logical shift,
  double 2-14, 10-27
  wide 2-14, 10-155
Logical skip instructions, fixed-point 2-14
Logical skip on condition instructions 2-14
Logical word address 8-7
Long displacement 1-11
Long load CPU identification 8-12, 10-67
Loop during protection violation, infinite 5-16
Lower 128 Kbyte, restricting vector stack to 7-9
Lower stack limit 4-2
Lower-numbered segment 4-3
LPEF 2-21, 4-5, 4-8, 10-81
LPEFB 2-21, 4-5, 4-8, 10-81
LPHY 7-2f, 10-82
LPSHJ 4-5, 4-8, 5-5f, 10-82
LPSHJ, pushing with 4-5
LPSR 2-9f, 10-83
LPTE 8-5, 10-83
LRB 2-13, 10-83
LSBRA 8-2, 10-84
LSBRS 8-2, 10-84f
LSH 2-14, 10-85
LSN 9-8, 10-85f
LSTB 2-20, 10-86
LWADD 2-4, 10-86
LWADI 2-4, 10-86
LWDIV 2-5, 10-87
LWDO 5-3, 10-87
LWDSZ 2-9, 10-87
LWISZ 2-9, 10-88
LWLDA 2-3, 10-88
LWMUL 2-5, 10-88
LWSBI 2-4, 10-88
LWSTA 2-3, 10-89
LWSUB 2-4, 10-89

M

Machine status, testing 5-2
Magnitude E-3
Maintaining excess 64 notation 3-7f
Maintaining I/O facilities 7-5
Management instructions,
  C/350 program flow 9-10
  C/350 stack 9-11
Management,
  device 1-5, 7-1
  memory 1-6, 8-1
  program flow 1-5, 5-1
  queue 1-5, 6-1
  stack 1-4
  system 1-6, 8-1
Managing stack operations 4-1
Manipulating a device flag 7-5
Manipulating an interrupt mask 7-5
Manipulating an interrupt on flag 7-5
Manipulating modified flag 8-12
Manipulating referenced flag 8-12
Manipulating string of bytes 2-21
Manipulation instructions, PSR 2-9
Manipulation, byte 2-1
Mantissa 3-2, 3-8
Mantissa equals zero 3-1
Mantissa overflow 3-5f
Mantissa overflow flag 3-13
Mantissa sign, complementing the 3-7
Mantissa status 3-12f
Mantissa,
  adding one to the intermediate 3-6
  aligning 3-4f
  final intermediate 3-7
  intermediate 3-5ff
  normalizing intermediate 3-5ff
  range of 3-2
  range of the intermediate 3-5
  rounding an intermediate 3-5ff
  rounding down an even 3-5
  rounding the 3-4
  rounding up an odd 3-5
  shifting intermediate 3-5f
  truncating 3-4
  truncating intermediate 3-6ff
Map register assignments and format 7-2
Map, device 7-2f, 10-154
Mapped memory addressing 1-5
Mapped mode 7-2
Mapping and demand paging E-1
Maps,
  disabling data channel and BMC 7-3
  I/O instructions for BMC 7-3
  I/O instructions for data channel 7-3
Mask bit 7-12
Mask out instruction 7-6, 7-12
Mask skip and store if equal, wide 6-5, 10-157
Mask,
  changing interrupt 7-6
  current interrupt 7-12f
  enabling OVK 4-8
  fault service 1-3
  fixed-point overflow 2-10
  FTE 3-13
  initializing OVK 5-19, 7-13
  interrupt 7-5f, 7-12f
  I/O channel 7-12
  loading current interrupt 7-12
  manipulating an interrupt 7-5
  overflow fault service 1-2
  OVK 2-10, 4-8, 5-19, 5-25, 7-13, 9-10f
  resetting OVK 2-9
  resetting TE 3-12
  setting fixed-point overflow fault 5-19
  setting floating-point fault 5-20
  setting OVK 2-9
  setting TE 3-12f, 5-20f
  trap enable 3-12f, 5-19
  wide save and reset overflow 4-5, 5-5, 10-164f
  wide save and set overflow 4-5, 5-5, 10-165
  wide special save and reset overflow 4-5, 5-5, 9-3, 9-10f, 10-170f
  wide special save and set overflow 4-5, 5-5, 9-3, 9-10f, 10-171
Maximum number of gates 5-10f
Memory access restrictions 1-6
Memory address space 1-1
Memory address translation 8-2f, 8-7
Memory addressing 1-5, 1-8, 9-10
Memory and device, transfers between 7-1ff
Memory location,
  narrow skip on all bits set in 10-99
  narrow skip on any bit set in 10-99
  reserved 5-15
  updating WSP reserved 5-25
  wide skip on all bits set in 10-163
  wide skip on any bit set in 10-164
Memory management 1-6, 8-1
Memory operand 1-12
Memory page, disk resident 8-1
Memory pages, swapping 8-8
Memory read operation 8-6
Memory reference instruction 1-8
Memory reference instruction,
  aborting 8-2
  C/350 9-1, 9-4ff
  resume 8-5
Memory reference instructions, C/350 9-4ff
Memory reference,
  invalid 5-14ff
  I/O 8-12

Memory resident flag 8-5
Memory resident page 8-5f, 8-12
Memory segment, defined 1-4
Memory size 8-12
Memory space, physical 8-11
Memory word to accumulator,
  narrow add 2-4, 10-78, 10-184
  wide add 2-4, 10-86, 10-191
Memory word,
  narrow add accumulator to 2-4
  narrow divide 2-5, 10-78, 10-185
  narrow multiply 2-5, 10-80, 10-187
  narrow subtract 10-81, 10-187
  subtract 2-4
  wide divide 2-5, 10-87, 10-191
  wide multiply 2-5, 10-88, 10-193
  wide subtract 10-89, 10-194
Memory words, reserved 5-25
Memory write operation 8-6
Memory,
  accessing 1-8, 8-2
  convert FPAC data and load in 2-20
  convert the four FPAC's and load in 2-20
  defined virtual 1-6
  fix to 9-9, 10-46
  float from 9-9, 10-48
  loading byte in 2-20
  logical 8-1
  moving bytes in 2-15
  narrow skip on all bits set in 2-8
  narrow skip on any bit set in 2-8
  page table in 8-1
  physical 8-1, 8-8
  reserved 2-23, 4-1, 5-11, 5-13, 5-19ff, 7-9, 7-11, 8-8, 8-11, 9-2
  restoring page from 8-11
  virtual 8-1
  wide skip on all bits set in 2-8
  wide skip on any bit set in 2-8
  writing to 8-8
Microcode revision level 8-12
Microinterrupt D-2
Microinterrupt or fault 5-26
Microinterrupt return block fault code, invalid D-1
Microinterrupt return block protection violation, invalid 5-17
Mixed number to fraction, converting 3-3
Mnemonic, device address 7-4
Mode,
  defined I/O 7-1ff
  defined LEF 7-1ff, 8-3
  mapped or unmapped 7-2
Modified and referenced bits,
  load 8-12, 10-77
  store 8-12, 10-110
Modified bits, table of referenced and 8-1
Modified flag 8-12
Modify stack pointer 9-11, 10-91
Modify stack pointer, wide 4-4, 5-26, 9-11, 10-158
Modifying current segment field 1-5
Modifying guard digit 3-4
Modifying segment base register 8-2
Modifying stack register contents 4-2
Modifying WSP register 4-4
MOF flag 3-13, 5-20
Most significant bit 2-7
Most significant bit, defined 1-2
Most significant digit 2-17
MOV 2-3, 2-6, 2-8, 10-89f
Move alphabetics 10-27
Move and convert instructions, decimal 2-20
Move and skip 2-3
Move characters 10-28
Move decimal/floating-point instructions 2-20
Move digit with overpunch 2-21, 10-29
Move float 10-28
Move floating-point 10-50
Move instruction, character 5-12
Move instructions,
  byte 2-20f
  decimal 2-20f
  fixed-point 2-3
  floating-point 3-3f
Move j alphabetical characters 2-21
Move j characters 2-21
Move j float 2-21
Move j numerics 2-21
Move numeric with zero suppression 2-21, 10-30
Move numerics 10-29
Move right, wide 2-21, 10-158
Move until true,
  character 9-8, 10-13f
  wide character 2-20, 9-8, 10-126f
Move with optional skip 2-6, 2-8, 10-89f
Move,
  block 9-8, 10-10
  block add and 9-8, 10-8f
  character 9-8, 10-15f
  wide 2-3, 10-158
  wide block 2-3, 9-8, 10-121f
  wide character 2-20, 9-8, 10-127f
  wide decimal 2-20, 10-132f
Movement instructions,
  fixed-point byte 2-20
  fixed-point data 2-3
Moving bytes 2-15, 2-20f
Moving decimal numbers 2-15
Moving floating-point data 3-4
MSKO 7-6, 7-12
MSP 9-11, 10-91
MUL 2-5, 10-91
MULS 2-5, 10-92

Multi-word wide stack instructions 4-8
Multiple accumulators,
  pop 9-8, 9-11, 10-102, 10-160
  push 9-8, 9-11, 10-103f
  wide pop 2-3, 4-1, 4-5, 4-8, 9-8, 9-11, 10-160
Multiple protection violations 5-14ff
Multiplexor channel I/O, burst 1-5f, 7-1f
Multiplication instructions, fixed-point 2-5
Multiplication, floating-point 3-7
Multiply double (FPAC by FPAC) 3-7, 10-49f
Multiply double (FPAC by memory) 3-7, 9-9, 10-49, 10-74f, 10-181
Multiply memory word,
  narrow 2-5, 10-80, 10-187
  wide 2-5, 10-88, 10-193
Multiply single (FPAC by FPAC) 3-7, 10-50
Multiply single (FPAC by memory) 3-7, 9-9, 10-49, 10-74f, 10-182
Multiply,
  narrow 10-98
  narrow sign extend 2-5
  signed 2-5, 10-92
  unsigned 2-5, 10-91
  wide 2-5, 10-159
  wide signed 2-5, 10-159

N

N flag (negative flag) 3-8, 3-13
N flag of the FPSR 3-10
N variable in PC relative addressing 1-11
NADD 2-4, 10-92
NADDI 2-4, 10-92
NADI 2-4, 10-93
Narrow add 2-4, 10-92
Narrow add accumulator to memory word 2-4
Narrow add immediate 2-4, 10-78, 10-93, 10-185
Narrow add memory word to accumulator 2-4, 10-78, 10-184
Narrow backward search queue and skip 5-3, 6-5, 10-93f
Narrow data, computing 2-2
Narrow decrement and skip if zero 2-9, 9-8, 10-79, 10-186
Narrow divide 10-94
Narrow divide memory word 2-5, 10-78, 10-185
Narrow DO until greater than 10-79, 10-185f
Narrow extended add immediate 2-4, 10-92
Narrow fault return block 5-24
Narrow floating-point fault return block 5-20
Narrow floating-point fault, servicing 5-20
Narrow forward search queue and skip 5-3, 6-5, 10-96f
Narrow frame pointer 9-2
Narrow halve (AC/2) 2-5, 10-61
Narrow immediate,
  wide add with 10-159
  wide arithmetic shift with 10-121
  wide logical shift with 10-156
Narrow increment and skip if zero 2-9, 9-8, 10-79, 10-186
Narrow load accumulator 2-3, 9-8, 10-80, 10-186
Narrow load CPU identification 8-12, 10-94
Narrow load immediate 2-3, 10-98
Narrow multiply 10-98
Narrow multiply memory word 2-5, 10-80, 10-187
Narrow negate 2-13, 10-98
Narrow queue, searching 6-5
Narrow return address 5-27
Narrow return block 9-2
Narrow return block for decimal data, type 1 fault 5-24
Narrow return block, pushing 5-20f, 5-27
Narrow sign extend divide or multiply 2-5
Narrow skip on all bits set in accumulator 2-8, 10-98
Narrow skip on all bits set in memory 2-8, 10-99
Narrow skip on any bit set in accumulator 2-8, 10-99
Narrow skip on any bit set in memory 2-8, 10-99
Narrow stack 1-4, 4-1, 5-20, 9-2, 9-4
Narrow stack fault 5-25
Narrow stack fault handler 5-27f
Narrow stack fault operations 5-26
Narrow stack fault return block 5-28
Narrow stack fault, servicing 5-25, 5-27
Narrow stack limit 5-26ff, 9-2
Narrow stack management 1-4
Narrow stack operations, managing 4-1
Narrow stack overflow fault 5-26f
Narrow stack parameters 9-2
Narrow stack pointer 5-26ff, 9-2
Narrow stack underflow 5-27
Narrow stack underflow fault 5-26, 9-2
Narrow stack,
  defined 5-25
  defining limits of 9-2
  initializing 5-27
  location of 5-27
  pushing and popping 5-26
Narrow store accumulator 2-3, 9-8, 10-80, 10-187
Narrow subtract 2-4, 10-100
Narrow subtract immediate 10-80, 10-100, 10-187
Narrow subtract memory word 10-81, 10-187
Natural logarithm double, floating-point 10-142
Natural logarithm single, floating-point 10-142
Natural logarithm, floating-point 3-11
NBStc 5-3f, 6-5, 10-93f
NCLID 8-12, 10-94
NDIV 2-5, 10-94
NEG 2-6, 2-13f, 10-95f
Negate 2-13, 3-3, 10-50, 10-95f

Negate,
narrow 2-13, 10-98
optional shift logical 2-14
wide 2-13, 10-159
with optional skip 2-6, 2-13f
Negative number, shifting a 2-6
Never, skip 5-3
Next instruction, addressing 5-1
NFStc 5-3f, 6-5, 10-96f
NIO 7-3, 10-97, A-2
NLDAI 2-3, 10-98
NMUL 2-5, 10-98
NNEG 2-13, 10-98
No error, skip on (ANY = 0) 3-9, 10-57
No invalid input argument,
skip on (INV = 0) 10-57
skip on no overflow and (OVF and INV = 0) 10-58
skip on no underflow and (UNF and INV = 0) 10-58
No I/O transfer 7-3, 10-97
No mantissa overflow, skip on (MOF = 0) 3-9, 10-57
No overflow and no invalid input argument, skip on (OVF and INV = 0) 10-58
No overflow and no zero divide, skip on (OVF and INV = 0) 3-9
No overflow,
skip on (OVF = 0) 3-9, 10-57
skip on no underflow and (UNF and OVF = 0) 3-9, 10-58
No skip 5-3, 10-51
No underflow and no invalid input argument, skip on (UNF and INV = 0) 10-58
No underflow and no overflow, skip on (UNF and OVF = 0) 3-9, 10-58
No underflow and no zero divide, skip on (UNF and INV = 0) 3-9
No underflow, skip on (UNF = 0) 3-9, 10-58
No zero divide,
skip on (INV = 0) 3-9
skip on no overflow and (OVF and INV = 0) 3-9
skip on no underflow and (UNF and INV = 0) 3-9
Noninterruptible instructions 7-6
Nonprivileged fault 1-16f, 5-13
Nonprivileged fault pointer 5-13
Nonprivileged fault, servicing 5-14
Nonresident page 1-16
Nonresident page fault 5-13
Nonsign-positioned numbers, for unpacked decimal 2-19f
Nonzero bit,
skip on 9-6ff
wide skip on 2-8, 2-14
Nonzero,
decrement and jump if 10-21
skip on (Z = 0) 3-9, 10-57
Normal program execution 5-19f
Normal program flow 5-1
Normalize 10-50
Normalize, floating-point 3-3
Normalized format E-3
Normalizing 3-4
Normalizing floating-point data 3-1
Normalizing intermediate mantissa 3-5ff
Normalizing the result 3-5
Notation, maintaining excess 64 3-7f
NSALA 2-8, 10-98
NSALM 2-8, 10-99
NSANA 2-8, 10-99
NSANM 2-8, 10-99
NSBI 2-4, 10-100
NSUB 2-4, 10-100
Number combination, for unpacked decimal sign and 2-19f
Number of arguments 5-12f
Number of words required beyond WSL 4-8
Number too large to convert D-2
Number,
interrupting device 7-12f
physical page 7-2
Numbers, compare two floating-point (set N and Z) 3-9
Numeric with zero suppression, move 2-21, 10-30
Numerical algorithms, floating-point A-2
Numerics,
move 10-29
move j 2-21

O

Offset,
page 8-8
page table 8-3
program counter 5-10ff
termination 5-3
word 1-15, 9-6
Once, insert character 2-21, 10-25
One to the intermediate mantissa, adding 3-6
One,
set bit to 9-6f, 10-10
set carry to 10-18
set T to 10-34
skip on zero bit and set bit to 9-6, 9-8, 10-117
wide set bit to 9-8, 10-123f
wide skip on accumulator bit set to 10-168
wide skip on zero bit and set bit to 9-8, 10-174
One's complement with optional skip 2-6, 2-14
One's complement, optional shift logical 2-14

One-element queue 6-3
One-level page table 8-2f, 8-7ff
Opcode pointer if sign flag is zero 2-21
Opcode pointer if trigger is one 2-21
Opcode violation, unimplemented 5-15
Opcode,
  invalid 2-24, 5-22, D-2
  I/O instruction 7-4
  unimplemented 5-14ff
  valid instruction 5-19
Opcodes,
  interpreting LEF and I/O 7-1ff
  wide instruction A-1
Operand 1-12
Operand,
  bit 1-15
  byte 1-13
  double word 1-2, 1-13
  memory 1-12
  size 1-8
  word 1-2, 1-13
Operands, signs of the two 3-6
Operating system, kernel 1-6
Operation,
  extended 4-5, 4-8, 9-10f, 10-103, 10-188
  finishing an I/O 7-5
  illegal I/O 8-3
  initiating I/O 7-5
  I/O instruction 7-4
  legal I/O 8-3
  memory read 8-6
  memory write 8-6
  overflow stack 5-26, D-2
  read 8-12
  stack 1-16
  stack fault 5-13
  testing the results of an 2-7
  underflow stack 5-26
  wide extended 5-5, 9-10f, 10-176
  wide pop 5-25
  wide stack example 5-7f
  write 8-12
Operations,
  binary 2-1
  byte 2-15
  checking for valid 5-13
  decimal and byte 2-15
  decimal arithmetic 2-15
  device independent 7-3
  fixed-point logical 2-12
  floating-point arithmetic 3-4
  logical 2-1
  managing narrow stack 4-1
  managing wide stack 4-1
  narrow stack fault 5-26
  shift 2-7
  stack 4-1
  wide stack 4-1
  wide stack fault 5-25
Option, skip 2-7
Optional device flag handling 7-3
OR immediate,
  exclusive 2-13, 10-189
  inclusive 2-13, 10-64
  wide exclusive 2-13, 10-177
  wide inclusive 2-12f, 10-152
OR referenced bits 8-12, 10-100f
OR,
  exclusive 2-12f, 10-188
  inclusive 2-12f, 10-64
  wide exclusive 2-13, 10-177
  wide inclusive 2-13, 10-152
Order of arguments 5-12
ORFB 8-12, 10-100f
Original descriptor 5-23f
Original source indicator 5-23f
Outward call fault code D-1
Outward call protection violation 5-17
Outward return 5-8
Outward subroutine call, illegal 5-9
Overflow condition,
  fixed-point 2-10
  stack 4-3, 5-12
Overflow detection, enabling vector stack 7-9
Overflow fault 2-8
Overflow fault mask, setting fixed-point 5-19
Overflow fault service mask 1-2
Overflow fault,
  checking for stack 4-3
  detecting an 2-10
  detecting floating-point 5-20
  detecting wide stack 5-25
  disabling narrow stack 5-27
  disabling wide stack 4-7, 5-25
  fixed-point 1-2, 2-8, 5-19
  ignoring floating-point 5-20
  initiating fixed-point 5-19
  narrow stack 5-26f
  servicing an 2-10
Overflow flag 2-8, 2-10, 5-25, 7-13
Overflow flag,
  exponent 3-12
  mantissa 3-13
  setting 2-8
  setting fixed-point 5-19
Overflow mask,
  fixed-point 2-10, 9-10f
  wide save and reset 4-5, 5-5, 10-164f
  wide save and set 4-5, 5-5, 10-165
  wide special save and reset 4-5, 5-5, 9-3, 9-10f, 10-170f
  wide special save and set 4-5, 5-5, 9-3, 9-10f, 10-171

Overflow stack operation 5-26, D-2
Overflow,
  checking for stack 5-14
  checking for vector stack 7-13
  exponent 3-6
  fixed-point 2-6
  ignoring fixed-point 5-19
  mantissa 3-5f
  stack 4-6f
Overpunch, move digit with 2-21, 10-29
Overwrite a page 8-1
Overwriting data in area beyond the stack 5-25
OVF flag 3-6, 3-12, 5-20
OVF flag of the FPSR 3-10
OVK fixed-point overflow mask 9-10f
OVK mask 2-10, 4-8, 5-19, 5-25, 7-13
OVK mask,
  initializing 5-19, 7-13
  resetting 2-9
  setting 2-9, 5-25
  wide save/reset 4-8
  wide save/set 4-8
  wide special save/reset 4-8
  wide special save/set 4-8
OVK to one,
  wide save and set 2-9f
  wide special save and set 2-9
OVK to zero,
  wide save and set 2-9
  wide special save and set 2-9
OVR flag 2-8, 2-10, 5-19, 5-25, 7-13
OVR flag,
  initializing 5-19, 7-13
  setting 5-25
  skip on reset 2-8, 10-111

P

P depending on S, add to 10-20
P depending on T, add to 10-21
P,
  add to 10-21
  current value of 5-23f
Packed byte 2-15
Packed decimal 2-15
Packed decimal data, converting 2-20
Packed decimal string 2-17
Page 8-8, E-1
Page access 8-5, 8-8
Page access flag, valid 8-5
Page access,
  type of 8-5
  valid 8-5
Page address,
  physical 8-7
  shifted 8-4
Page fault 8-5, 8-8, 8-11
Page fault handler 5-13, 8-11f
Page fault,
  cause of 8-11
  nonresident 5-13
  servicing 8-5, 8-11
Page from disk, loading referenced 8-11
Page from memory, restoring 8-11
Page level one 8-8
Page level two 8-8
Page number, physical 7-2
Page offset 8-8
Page protocol access, ignoring 8-7f
Page protocols 1-8f
Page table E-1
Page table depth fault 8-7
Page table entries 8-1
Page table entry 8-7f
Page table entry,
  format 8-5
  load 8-5, 10-83
  store 8-5, 10-111
Page table frame (frame number), root 8-3
Page table frame, root 8-2
Page table in memory 8-1
Page table logical word address, one-level or two-level 8-7
Page table offset 8-3
Page table page, addressing another 8-7
Page table translation,
  example of one-level 8-9
  example of two-level 8-10
  one-level 8-8
  two-level 8-7f
Page table validity protection fault 8-8
Page table validity violation 5-15
Page table,
  accessing 8-5
  address of first entry in 8-2ff
  one-level 8-2f, 8-7ff
  two-level 8-2f
Page tables 8-1, 8-5
Page word offset (see pageframe) 8-4
Page zero 4-2, 5-10f, 5-13, 5-25, 5-27f, 8-11
Page,
  accessing physical 8-2
  addressing another page table 8-7
  defined 8-1
  disk resident 8-5
  disk resident memory 8-1
  invalid 8-5
  location 8-1
  memory resident 8-5f, 8-12
  nonresident 1-16
  overwrite a 8-1

physical 8-12
referenced 8-1
restricted 8-8
valid 8-5, 8-8
valid referenced 8-8
Pageframe format 8-4
Pageframes 8-3
Pages,
defined 1-7
frequency of references to 8-12
status of 8-5
swapping memory 8-8
Paging,
defined demand 8-11
mapping and demand E-1
Parameter,
vector stack fault address 7-9
vector stack limit 7-9
vector stack pointer 7-9
Parameters of I/O data transfer 7-2
Parameters,
managing stack 4-1
narrow stack 9-2
pushing previously saved stack 7-9
saving wide stack 7-9
storing processor 8-12
vector stack 7-9
wide stack 4-1
Passing arguments 5-12
Passing arguments to a subroutine 5-7
PATU 8-12, 10-101
PBX 2-9, 4-5, 4-8, 5-5f, 10-101
PC 5-19, 5-23, 5-26, 5-28, 9-4, D-2
PC and jump,
pop 9-10f, 10-103
wide pop 4-5, 4-8, 5-5, 9-10f, 10-161
PC relative address mode 1-12
PC relative addressing, n variable in 1-11
PC relative skip 5-3
PC return address 4-6
Performing floating-point computations 2-15
Physical address 1-7, 1-11, 7-2, 8-1ff
Physical address instruction, load 7-2
Physical address into accumulator, loading 7-3
Physical address to device, sending 7-2
Physical address translation 8-3
Physical address,
forming 7-2
logical address to 8-1
Physical and conditional skip, load 10-82
Physical memory 8-1, 8-8
Physical page 8-12
Physical page address 8-7
Physical page number 7-2
Physical page, accessing 8-2
PIO 7-3, 10-102
Point location, binary 3-2
Point to a segment, entry 5-9f
Pointer 5-20, 7-6, 8-11
Pointer chain 7-13
Pointer contents, store accumulator in stack 10-113
Pointer format, byte 1-14
Pointer if sign flag is zero, opcode 2-21
Pointer if trigger is one, opcode 2-21
Pointer parameter, vector stack 7-9
Pointer to fault subopcode, byte 5-24
Pointer to subopcode causing fault, byte 5-23
Pointer to wide stack fault handler, saving 7-9
Pointer to word pointer, converting byte 2-21
Pointer wraparound, narrow stack 5-27
Pointer,
bit 1-15
bit zero of narrow stack 5-27
bit zero of wide stack 5-26
byte 1-13f, 5-12, 9-5f
comparing wide stack 4-3
converting byte pointer to word 2-21
decrementing wide stack 4-3f
defined 1-11
forming bit 9-6
frame 9-2
incrementing wide stack 4-3f
indirect 1-11, 2-23, 5-10, 5-13, 5-19ff, 5-25, 5-28, 9-7, 9-9ff
initializing wide frame 4-3, 5-11
initializing wide stack 4-3, 8-11
initializing wide stack fault 7-9
loading wide frame 5-12f
loading wide stack 5-11, 5-13
location bit 9-6
modify stack 9-11, 10-91
narrow frame 9-2
narrow stack 5-26ff, 9-2
nonprivileged fault 5-13
privileged fault 5-13
protecting against trojan horse 5-12
saving wide frame 7-9
saving wide stack 7-9, 7-11
setting bit zero of narrow stack 5-27f
setting narrow stack 5-27
setting wide stack 4-3
single word 9-5f
single word indirect 9-7
skip on valid byte 10-117
skip on valid word 10-118
storing wide frame 5-7, 5-11, 5-13, 8-11
storing wide stack 5-11, 5-13, 8-11
updating wide stack 5-25
using the wide frame 4-3
wide frame 4-2f, 5-6ff, 5-15, 7-9
wide modify stack 4-4, 5-26, 9-11, 10-158
wide stack 4-2f, 4-7, 5-12, 5-15, 5-25, 7-9

word 1-15, 5-12, 9-6
zero-extending vector stack 7-9
Pointers,
C/350 9-3
trojan horse 5-12
POP 9-8, 9-11, 10-102
Pop block and execute 2-9, 4-5, 4-8, 5-5f, 5-28, 9-10f, 10-101, 10-103
Pop block and execute, wide 2-9, 4-5, 4-8, 5-8, 5-14, 5-26, 9-10f, 10-160f
Pop block, wide 2-9, 4-5, 4-8, 5-5f, 5-8, 5-14, 5-26, 9-10f, 10-160f
Pop context block, wide 2-9, 8-8, 8-11f, 10-133
Pop floating-point state 3-12, 9-9, 10-51f
Pop floating-point status and accumulators 10-51f
Pop multiple accumulators 9-8, 9-11, 10-102, 10-160
Pop multiple accumulators, wide 2-3, 4-1, 4-5, 4-8, 9-8, 9-11, 10-160
Pop operation, wide 5-25
Pop PC and jump 9-10f, 10-103
Pop PC and jump, wide 4-5, 4-8, 5-5, 9-10f, 10-161
Pop, wide floating-point 3-4, 4-5, 4-8, 10-144f
POPB 5-28, 9-10f, 10-103
POPJ 9-10f, 10-103
Popping accumulators 4-1
Popping narrow stack 5-26
Popping return block 4-1, 4-5, 5-13
Popping wide return block 4-3, 5-4, 7-7
Popping with WPOP 4-5
Popping with WPOPJ 4-5
Port select instruction 7-2
Port, I/O (see I/O channel)
Power double, floating-point 10-147
Power fail flag 7-4f
Power failure, detecting 7-5
Power instruction error, floating-point 3-14
Power single, floating-point 10-147
Power up 2-10
Power voltage ranges, detecting proper 7-5
Power, floating-point 3-11
Powers of 2 table C-1
Precision conversion, fixed-point 2-2
Precision,
double 3-1f, 3-6f
fixed-point data 2-2
single 3-1f, 3-4, 3-6f
Previously saved stack parameters, pushing 7-9
Priority of handling faults 5-13
Priority of handling I/O interrupts 5-14
Privileged access fault 5-12
Privileged fault 1-16f
Privileged fault pointer 5-13
Privileged fault, detecting 5-13
Privileged instruction protection violation 5-17
Privileged instruction violation 5-15
Privileged instruction violation fault code D-1
Privileged instructions 8-2, 8-12
Privileges destination segment, access 5-12
Processing, interrupt 5-6, 5-8, 7-7, 7-9, 7-11
Processor identification instructions, central 8-12
Processor instruction, equivalent 32-bit 9-7
Processor parameters, storing 8-12
Processor state, internal 7-9
Processor status register 1-2, 2-9, 5-12f, 5-19f, 5-23, 5-26, 7-6, 9-1, 9-4
Processor status register format 2-10
Processor status register from AC0, store 10-111
Processor status register in AC0, load 10-83
Processor status register, setting 5-19, 5-21, 7-3
Processor storage capacity, exceeding 1-2
Processor,
defined 1-1
restoring state of 8-11
saving current state of 8-8, 8-11
state of the 8-11
Producing an intermediate exponent 3-8
Producing an intermediate mantissa 3-8
Program compatibility, upward 4-1, 9-1
Program control to another segment, transferring 5-8ff
Program control,
returning 5-5f, 5-13
to another segment transferring 5-8f
transferring 1-8, 4-2, 4-4, 5-6, 5-10, 5-21, 7-13
Program counter 1-5, 2-7, 2-22f, 5-1ff, 5-11f, 5-21, 5-28, 7-13, 9-2
Program counter offset 5-10ff
Program counter wraparound A-1
Program counter,
floating-point 3-8, 3-10, 3-14
format 1-5, 9-2
incrementing 3-9, 5-1
jump relative to 9-10
loading 5-13
restoring 5-4
updating 5-14, 5-25, 5-28
Program development, supporting C/350 9-2
Program execution, normal 5-19f
Program flow 5-2, 5-4
Program flow instructions, C/350 9-2, 9-10
Program flow management 1-5, 5-1
Program flow management instructions, C/350 9-10
Program flow,
altering normal 5-1
normal 5-1
Program relative addressing 1-11
Program,
executing interrupted 5-14
expanding ECLIPSE C/350 9-3
restarting interrupted 8-11
Programmed I/O 1-5, 7-1, 10-102

Programmed I/O command, issuing 7-3
Programming, C/350 9-1
Programs, supporting 16-bit and 32-bit 4-1
Protecting against trojan horse pointer 5-12
Protection E-2
Protection capabilities, system 1-1
Protection fault 2-22, 5-11, 5-19, 8-7f
Protection fault code 8-8
Protection fault code, validity bit D-1
Protection fault codes 5-17
Protection fault handler 5-15f
Protection fault handler definition 5-15f
Protection fault,
  infinite 5-15
  initiating a 5-8
  page table validity 8-8
  segment validity 8-2
Protection mechanism,
  accessing 1-17
  changing 1-17
  hierarchical 1-17
Protection violation 1-16, 8-2f, 8-5f
Protection violation fault codes D-1
Protection violation faults 1-8f, 5-13
Protection violation priorities 5-15
Protection violation priority levels 5-15
Protection violation sequence 5-18
Protection violation,
  defer (indirect) 5-17
  detecting 7-2
  execute 5-17
  I/O 5-17
  I/O interrupt during 5-17
  illegal gate 5-17
  indirect addressing 1-11
  infinite loop during 5-16
  invalid microinterrupt return block 5-17
  inward address reference 5-17
  inward return 5-17
  outward call 5-17
  privileged instruction 5-17
  read 5-17
  unimplemented instruction 5-17
  validity 5-17
  write 5-17
Protection violations 5-14ff
Protection violations, multiple 5-14ff
Protocols, page and segment 1-8f
PRTSEL 7-2
PSH 9-8, 9-11, 10-103f
PSHJ 9-10f, 10-104
PSHR 9-10f, 10-104
PSR 4-6, 5-6f, 5-19f, 5-23, 5-26, 7-12, 9-3
PSR from AC0, store 2-9
PSR in AC0, load 2-9f
PSR manipulation instructions 2-9
PSR word in device control table 7-13
PSR,
  setting the 5-15
  storing 5-7
PTE 8-5
Purging address translator 8-12, 10-101
Push accumulators, wide 2-3, 4-5, 4-8, 9-8, 10-161
Push address 2-21, 4-8, 10-81, 10-189
Push and jump 4-8
Push and jump to subroutine 4-5, 4-8, 5-6
Push byte address 2-21
Push effective address 2-21
Push effective byte address 2-20f, 4-5f, 4-8, 10-81, 10-189
Push floating-point state 3-12, 9-9, 10-52f
Push floating-point status and accumulators 10-52f
Push jump 5-5, 9-10f, 10-82, 10-104, 10-189
Push multiple accumulators 9-8, 9-11, 10-103f
Push return address 5-6, 9-10f, 10-104
Push, wide floating-point 3-4, 4-5, 4-8, 10-146
Pushed, number of arguments 5-12
Pushing a byte address 4-5
Pushing a double word 4-3
Pushing a word address 4-5
Pushing accumulators 4-1
Pushing arguments onto a wide stack 5-7
Pushing byte address 4-5
Pushing double word onto vector stack 7-12
Pushing fault return block 5-14
Pushing narrow fault return block 5-24
Pushing narrow return block 5-20f, 5-27
Pushing narrow stack 5-26
Pushing previously saved stack parameters 7-9
Pushing return block 4-1, 4-5, 5-13, 5-21, 9-3
Pushing wide return block 5-4, 5-6, 5-12, 5-19ff
Pushing wide return block onto vector stack 7-9, 7-11
Pushing with LPSHJ 4-5
Pushing with WPSH 4-5
Pushing with XPSHJ 4-5
Pushing word address 4-5

Q

Queue and skip,
  narrow backward search 5-3, 6-5, 10-93f
  narrow forward search 5-3, 6-5, 10-96f
  wide backward search 5-3, 6-5, 10-122f
  wide forward search 5-3f, 6-5, 10-150
Queue data element, dequeue a 10-22
Queue descriptor 6-2
Queue element 5-4
Queue instructions 6-5
Queue instructions, search 5-4
Queue link 6-2
Queue management 1-5, 6-1
  7-11, 8-8, 8-11, 9-2

Reserved memory location 5-15
Reserved memory location, updating WSP 5-25
Reserved memory words 5-25
Reset lead bit,
  locate and 2-13, 10-83
  wide locate and 2-13, 10-155
Reset overflow mask,
  wide save and 4-5, 5-5, 10-164f
  wide special save and 4-5, 5-5, 9-3, 9-10f, 10-170f
Reset referenced bits 8-12, 10-104f
Reset,
  I/O 2-10, 7-3
  skip on OVR flag 2-8, 10-111
  system 2-10
Resetting OVK mask 2-9
Resetting TE mask 3-12
Resident flag, memory 8-5
Resident memory page, disk 8-1
Resident page,
  disk 8-5
  memory 8-5f, 8-12
Responding to interrupt request 7-6
Restarting interrupted instruction 7-6
Restarting interrupted program 8-11
Restore 9-10f, 10-105
Restore from an I/O interrupt, wide 4-5, 5-8, 7-7
Restore, wide 2-9, 4-8, 9-10f, 10-162
Restoring accumulators 5-4
Restoring carry 5-4
Restoring page from memory 8-11
Restoring program counter 5-4
Restoring state of processor 8-11
Restricted page 8-8
Restricting vector stack to lower 128 Kbyte 7-9
Restrictions, memory access 1-6
Result for a condition, testing 3-8
Result,
  calculating the 3-5
  intermediate 3-6
  normalizing the 3-5
  rounding the 3-5
  rounding the intermediate 3-13
  shifting an intermediate 2-6
  signs of the intermediate 3-5
  storing the floating-point 3-6
  truncating intermediate 3-13
  truncating the 3-5
Results of an operation, testing the 2-7
Results, shifting decimal 2-22
Resume flag, interrupt (see IRES flag)
Resume memory reference instruction 8-5
Resuming interrupted instruction 7-6
Retrieving data from the wide stack 4-3
Return 9-4, 9-10f, 10-106
Return address 5-2, 5-7, 5-13, 5-19f, 5-25
Return address,
  narrow 5-27
  PC 4-6
  push 5-6, 9-10f, 10-104
  storing 5-7, 7-7
Return block 4-4, 5-12, 5-19f, 5-25f, 5-28, 9-10f, D-2
Return block fault code, invalid microinterrupt D-1
Return block for ASCII data, type 2 or 3 fault 5-23
Return block for decimal data,
  type 1 fault narrow 5-24
  type 1 fault wide 5-23
Return block instructions, wide stack 4-5
Return block onto vector stack, pushing wide 7-9, 7-11
Return block protection violation, invalid microinterrupt 5-17
Return block type 2-24, 5-22
Return block,
  C/350 9-2
  defined wide 5-6f
  fault 5-15f
  fixed-point fault 5-19
  floating-point fault 5-20
  narrow 9-2
  narrow floating-point fault 5-20
  narrow stack fault 5-28
  popping 4-1, 4-5, 5-13
  popping wide 4-3, 5-4, 7-7
  pushing 4-1, 4-5, 5-13, 5-21, 9-3
  pushing fault 5-14
  pushing narrow 5-20f, 5-27
  pushing narrow fault 5-24
  pushing wide 5-4, 5-6, 5-12, 5-19ff
  size 5-21
  stack fault 4-8, 5-26, 5-28
  standard wide 4-6
  wide 4-3, 5-7, 5-25
  wide floating-point fault 5-20
  wide stack fault 5-26
Return blocks,
  types of narrow fault 5-24
  types of wide fault 5-23
Return instruction,
  executing 5-1
  wide 5-6f, 5-13, 9-3f
Return via wide save, wide 4-5
Return,
  illegal inward 5-9
  outward 5-8
  subroutine 5-4, 5-13
  wide 2-9, 4-8, 5-5, 5-8, 9-3f, 9-10f, 10-163
Returning from breakpoint handler 4-5
Returning from fault handler 3-13, 4-1
Returning from I/O interrupt 4-1, 5-6, 5-8

Queue,
building a 6-1
defined 1-5, 6-1
empty 6-3
one-element 6-3
ready 6-1
searching 6-5

R

Range of displacement 1-11
Range of mantissa 3-2
Range of the intermediate mantissa 3-5
Range of the logical address, checking 8-7
Range,
addressing 1-10, 8-7, 9-10
valid addressing 8-3
Ranges,
address 1-10, 1-12, 8-7, 9-10
detecting proper power voltage 7-5
Rates, data transfer 7-1
Read access 1-9
Read access flag 8-6
Read access violation 5-15
Read access, valid 8-6
Read command to device map, issuing 7-3
Read data, accessing page to 8-5
Read high word 10-54
Read operation 8-12
Read operation, memory 8-6
Read protection violation 5-17
Read violation fault code D-1
Read, valid 8-8
Reading high floating-point word 3-3
Ready queue 6-1
Recognition,
disabling I/O interrupt 7-5, 7-12
enabling fixed-point fault 2-10, 5-14
enabling floating-point fault 3-11ff, 5-20
enabling I/O interrupt 7-5, 7-12
enabling stack fault 5-25
Redefining wide stack 5-10f, 5-13
Redefining wide stack for segment zero 8-11
Reference violation, inward 5-15
Reference,
I/O memory 8-12
valid segment 8-7
Referenced and modified bits, table of 8-1
Referenced bits,
load modified and 8-12, 10-77
OR 8-12, 10-100f
reset 8-12, 10-104f
store modified and 8-12, 10-110
Referenced flag 8-12
Referenced page 8-1
Referenced page from disk, loading 8-11
Referenced page, valid 8-8
Register assignments and format, map 7-2
Register contents, modifying stack 4-2
Register format, processor status 2-10
Register initialization, stack 5-15
Register instructions, wide stack 4-4
Register,
bit zero of segment base 8-2
floating-point status 1-3, 3-4, 3-6, 3-12f
initializing wide stack 4-2, 4-4
load floating-point status 9-9, 10-48
loading segment base 8-2
modifying segment base 8-2
processor status 1-2, 2-9, 5-12f, 5-19f, 5-23, 5-26, 7-6, 9-1, 9-4
segment base 5-11, 7-1, 8-1f, 8-7
setting processor status 5-19, 5-21, 7-3
store floating-point status 9-9, 10-59, 10-75
testing the floating-point status 3-12
Registers,
C/350 9-1
defined wide stack management 1-4
fixed-point 1-2
initializing wide stack 7-9, 7-11
load all segment base 10-84
map 7-2
saving wide stack 7-9
specifying eight segment base 8-7
stack 4-1
Relative address mode, AC and PC 1-12
Relative address, effective or indirect 1-11
Relative addressing 1-10f, 1-13
Relative addressing,
C/350 accumulator 9-1
n variable in PC 1-11
program 1-11
Relative skip, PC 5-3
Relative to program counter, jump 9-10
Removing guard digit 3-5
Replacing SAVE/RTN with WSSVR/WRTN 9-4
Replacing SAVE/RTN with WSSVS/WRTN 9-4
Representation, excess 64 3-2
Representing decimal digit and sign 2-17
Request,
access 8-8
blocking an interrupt 7-5f
detecting an I/O interrupt 5-1
ignoring an interrupt 7-5
interrupt 7-6
I/O interrupt 2-10, 4-1
responding to interrupt 7-6
servicing I/O interrupt 7-5
Requesting I/O data transfer 7-5
Required beyond WSL for stack fault, words 4-8
Required beyond WSL, number of words 4-8
Reserved memory 2-23, 4-1, 5-11, 5-13, 5-19ff, 7-9,

Returning from LCALL 5-8
Returning from subroutine call 3-13, 4-1, 4-3, 4-5
Returning from XCALL 5-8
Returning program control 5-5f, 5-13
Returning with WPOPB 4-5
Revision level, microcode 8-12
Revision, floating-point 3-13
Right,
  DHXR double hex shift 2-22, 10-23f
  HXR single hex shift 2-22
  shifting one bit to the 2-7
  single hex shift 10-62
  wide move 2-21, 10-158
Ring, defined 1-6
RND 3-13
RND flag 3-4f, 3-13
Root evaluation, floating-point square 3-10f
Root page table frame 8-2
Root page table frame (frame number) 8-3
Round flag 3-4f, 3-13
Rounding 3-4
Rounding an intermediate mantissa 3-5ff
Rounding double to single, floating-point 3-3, 10-53f
Rounding down an even mantissa 3-5
Rounding the intermediate result 3-13
Rounding the mantissa 3-4
Rounding the result 3-5
Rounding up an odd mantissa 3-5
Routine dependent information, interrupt 7-12
Routine,
  device interrupt 7-12f
  interrupt service 7-5
  transferring to device interrupt 7-13
RRFB 8-12, 10-104f
RSTR 9-10f, 10-105
RTN 9-4, 9-10f, 10-106

S

S, add to P depending on 10-20
SAVE 5-26, 9-4, 9-10f, 10-106f, D-2
Save 9-4, 9-10f, 10-106f
Save and reset overflow mask,
  wide 4-5, 5-5, 10-164f
  wide special 4-5, 5-5, 9-3, 9-10f, 10-170f
Save and set overflow mask,
  wide 4-5, 5-5, 10-165
  wide special 4-5, 5-5, 9-3, 9-10f, 10-171
Save and set OVK to one,
  wide 2-9f
  wide special 2-9
Save and set OVK to zero,
  wide 2-9
  wide special 2-9
Save instruction,
  wide 4-3, 5-12
  wide special 5-5ff, 9-3f
Save without arguments 9-10f, 10-107f
Save, wide return via wide 4-5
Save/reset OVK mask,
  wide 4-8
  wide special 4-8
SAVE/RTN with WSSVR/WRTN, replacing 9-4
SAVE/RTN with WSSVS/WRTN, replacing 9-4
Save/set OVK mask,
  wide 4-8
  wide special 4-8
Saved stack parameters, pushing previously 7-9
Saving current state of processor 8-8, 8-11
Saving pointer to wide stack fault handler 7-9
Saving wide frame pointer 7-9
Saving wide stack parameters 7-9
Saving wide stack pointer 7-9, 7-11
Saving wide stack registers 7-9
SAVZ 9-10f, 10-107f
SBI 2-4, 10-108
SBR, setting bit two or three of 7-1ff
Scale factor field 2-16
Scaling floating-point number 3-3, 10-54
Scan until true instruction, wide character 2-23
Scan until true, wide character 10-128f
Scheme, addressing E-1
Search queue and skip,
  narrow backward 5-3, 6-5, 10-93f
  narrow forward 5-3, 6-5, 10-96f
  wide backward 5-3, 6-5, 10-122f
  wide forward 5-3f, 6-5, 10-150
Search queue instructions 5-4
Searches a string of bytes 2-23
Searching queue 6-5
Segment 4-1, 5-10, 8-5, 9-2, 9-4ff, 9-10f
Segment access 8-2
Segment access,
  type of 8-2
  valid 8-2
Segment base register 5-11, 7-1, 8-1f, 8-7
Segment base registers, load all 10-84
Segment call, too many arguments 5-26, D-2
Segment check, valid 8-3
Segment crossing 5-6, 5-8, 5-15, 5-26, D-2
Segment crossing violation 5-15
Segment field 1-8, 1-14f
Segment field, modifying current 1-5
Segment number 5-11
Segment of execution, changing current 7-6
Segment protocols 1-8f
Segment reference, valid 8-7
Segment transfer instructions 5-8
Segment validity flag 8-2
Segment validity protection fault 8-2
Segment validity violation 5-15

Segment zero 5-8, 5-13, 7-6f, 7-9, 7-11, 8-2, 8-11
Segment zero,
  crossing to 7-9, 7-11
  redefining wide stack for 8-11
Segment,
  access privileges destination 5-12
  accessing a 8-2
  accessing destination 5-9f
  another 4-2, 4-4, 5-6
  current 1-8, 4-1, 5-1, 5-6f, 5-13, 5-25, 5-28, 7-9, 7-11, 8-11, 9-2, 9-4ff
  defined 1-6
  defined current 1-4
  defined destination 1-9
  defined memory 1-4
  destination 5-6f, 5-10ff
  entry point to a 5-9f
  first 64 Kbytes of 9-2
  initiating a transfer to another 5-8
  invalid 8-2
  lower-numbered 4-3
  source 5-9ff
  transferring program control to another 5-8ff
  valid 8-3
Segmentation E-1
Segments, testing for identical 5-12
Select instruction, port 7-2
Selecting LEF mode 7-1ff
Sending physical address to device 7-2
Sequence of subroutine instructions 5-6
Sequence,
  calling 5-13
  DO-loop 5-3
  interrupt 7-8
Service mask,
  fault 1-3
  overflow fault 1-2
Service routine, interrupt 7-5
Service, concluding vector interrupt 7-11
Servicing a fault 1-2, 2-10, 3-13
Servicing a floating-point fault 3-12f
Servicing an interrupt 7-6
Servicing an interrupt request 7-5
Servicing an overflow fault 2-10
Servicing base level vector interrupt 7-9, 10-161
Servicing fault 5-25
Servicing floating-point fault 3-13, 5-20
Servicing intermediate level vector interrupt 7-11ff, 10-160f
Servicing narrow floating-point fault 5-20
Servicing narrow stack fault 5-25, 5-27
Servicing nonprivileged fault 5-14
Servicing page fault 8-5, 8-11
Servicing vector interrupt 7-3
Servicing wide floating-point fault 5-20
Servicing wide stack fault 5-25
Set bit to one 9-6ff, 10-10
Set bit to one,
  skip on zero bit and 9-6, 9-8, 10-117
  wide 9-8, 10-123f
  wide skip on zero bit and 9-8, 10-174
Set bit to zero 9-6ff, 10-10f
Set bit to zero, wide 9-8, 10-124
Set CARRY flag to one 2-6
Set CARRY flag to zero 2-6
Set carry to one 10-18
Set carry to zero 10-18
Set in accumulator,
  narrow skip on all bits 10-98
  narrow skip on any bit 10-99
  wide skip on all bits 10-163
  wide skip on any bit 10-164
Set in memory location,
  narrow skip on all bits 10-99
  narrow skip on any bit 10-99
  wide skip on all bits 10-163
  wide skip on any bit 10-164
Set overflow mask,
  wide save and 4-5, 5-5, 10-165
  wide special save and 4-5, 5-5, 9-3, 9-10f, 10-171
Set sign flag to one or zero 2-21, 10-33
Set T to one 10-34
Set T to zero 10-34
Set to one,
  bit 9-6ff
  wide skip on accumulator bit 10-168
Set to zero,
  bit 9-6ff
  wide skip on accumulator bit 10-168
Set trigger to one 2-21
Set trigger to zero 2-21
Setting an error flag 3-6
Setting bit two or three of SBR 7-1ff
Setting bit zero of narrow stack limit 5-27f
Setting bit zero of narrow stack pointer 5-27f
Setting bit zero of the WSL 5-25
Setting bit zero of the WSP 5-25
Setting BUSY and DONE flag 7-5
Setting fixed-point overflow fault mask 5-19
Setting fixed-point overflow flag 5-19
Setting floating-point fault flags 5-20
Setting floating-point fault mask 5-20
Setting INV flag 3-8
Setting ION flag 7-6
Setting IRES flag 5-25, 7-6
Setting modified flag 8-12
Setting N flag 3-8, 3-13
Setting narrow stack limit 5-27
Setting narrow stack pointer 5-27
Setting overflow flag 2-8
Setting OVK mask 2-9, 5-25

Setting OVR flag 5-25
Setting processor status register 5-19, 5-21, 7-3
Setting referenced flag 8-12
Setting RND flag 3-13
Setting TE mask 3-12f, 5-20f
Setting wide stack pointer 4-3
Setting WSP equal to WSL 5-25
Setting Z flag (true zero flag) 3-8, 3-13
SEX 2-2, 10-109
SGE 2-8, 10-109
SGT 2-8, 10-109
Shift immediate, wide logical 10-156
Shift instructions 2-6
Shift instructions,
  decimal 2-22
  hex 2-22
  logical 2-14
  wide arithmetic 2-6
Shift left immediate, wide logical 2-14
Shift left,
  DHXL double hex 2-22, 10-23
  HXL single hex 2-22
  single hex 10-61
Shift operations 2-7
Shift right,
  DHXR double hex 2-22, 10-23f
  HXR single hex 2-22
  single hex 10-62
Shift with narrow immediate,
  wide arithmetic 10-121
  wide logical 10-156
Shift,
  double logical 2-14, 10-27
  logical 2-14, 10-85
  wide arithmetic 10-120
  wide logical 2-14, 10-155
Shifting 0 to 31 bits 2-6
Shifting a negative number 2-6
Shifting an intermediate mantissa 3-5
Shifting an intermediate result 2-6
Shifting decimal results 2-22
Shifting one bit to the left 2-6
Shifting one bit to the right 2-7
Shifting the intermediate mantissa 3-6
SI, add to 10-21
Sign and number combination, for unpacked decimal 2-19f
Sign bit 3-2
Sign bit change 2-6
Sign bit, defined 2-2
Sign code, invalid 2-24, 5-22, D-2
Sign extend 16-bits to 32-bits 2-2, 10-109
Sign extend and divide 2-5, 10-27
Sign extend divide, narrow 2-5
Sign extend multiply, narrow 2-5
Sign flag is zero, opcode pointer if 2-21
Sign flag to one or zero, set 2-21, 10-33
Sign flag, insert character depending on 2-21
Sign magnitude data 3-1
Sign,
  complementing the mantissa 3-7
  insert 10-25
  load 9-8, 10-85f
  representing decimal digit and 2-17
  wide load 2-13, 2-23, 9-8, 10-156f
Sign-extending to 31 bits, displacement 1-11
Signed divide 2-5, 10-26
Signed divide, wide 2-5, 10-132
Signed divided instructions, C/350 A-2
Signed multiply 2-5, 10-92
Signed multiply, wide 2-5, 10-159
Signed skip if ACS greater than ACD, wide 2-8, 10-167
Signed skip if ACS greater than or equal to ACD, wide 2-8, 10-167
Signed skip if ACS less than ACD, wide 2-8
Signed skip if ACS less than or equal to ACD, wide 2-8, 10-168
Signed skip if less than, wide 10-169
Significant bit,
  least 2-6f
  most 2-7
Significant digit,
  least 2-17
  most 2-17
Signs of the intermediate result 3-5
Signs of the two operands 3-6
Sine double, floating-point 10-148
Sine single, floating-point 10-148
Sine, floating-point 3-11
Single hex shift left 10-61
Single hex shift left, HXL 2-22
Single hex shift right 10-62
Single hex shift right, HXR 2-22
Single precision 3-1f, 3-4, 3-6f
Single word indirect pointer 9-7
Single word pointer 9-5f
Single,
  add (FPAC to FPAC) 3-6, 10-43
  add (memory to FPAC) 3-6, 9-9, 10-42f, 10-72, 10-180
  divide (FPAC by FPAC) 3-8, 10-45
  divide (FPAC by memory) 3-8, 9-9, 10-44, 10-73, 10-180
  floating-point rounding double to 10-53f
  load floating-point 9-9, 10-47, 10-73, 10-181
  multiply (FPAC by FPAC) 3-7, 10-50
  multiply (FPAC by memory) 3-7, 9-9, 10-49, 10-74f, 10-182
  store floating-point 9-9, 10-60, 10-76, 10-183
  subtract (FPAC from FPAC) 3-7, 10-59
  subtract (memory from FPAC) 3-7, 9-9, 10-56, 10-75, 10-182

Size field 2-16
Size, memory 8-12
Skip always 10-54
Skip and store if equal, wide mask 6-5, 10-157
Skip if accumulator equal to immediate, wide 2-23, 10-166
Skip if accumulator greater than immediate,
  wide 2-8, 10-167
  wide unsigned 2-8, 10-174
Skip if accumulator less than or equal to immediate,
  wide 2-8, 10-169
  wide unsigned 2-8, 10-175
Skip if accumulator not equal to immediate, wide 2-8, 10-170
Skip if ACS equal to ACD, wide 2-8, 10-166
Skip if ACS greater than ACD 2-8, 10-109
Skip if ACS greater than ACD,
  wide signed 2-8, 10-167
  wide unsigned 2-8, 10-175
Skip if ACS greater than or equal to ACD 2-8, 10-109
Skip if ACS greater than or equal to ACD,
  wide signed 2-8, 10-167
  wide unsigned 2-8, 10-175
Skip if ACS less than ACD, wide signed 2-8
Skip if ACS less than or equal to ACD, wide signed 2-8, 10-168
Skip if ACS not equal to ACD, wide 2-8, 10-170
Skip if equal to immediate, wide 2-8
Skip if equal to, wide 2-23
Skip if less than, wide signed 10-169
Skip if zero 4-5
Skip if zero,
  decrement and 9-8, 10-34
  extended decrement and 9-8, 10-36
  extended increment and 9-8, 10-37
  increment and 9-8, 10-64
  narrow decrement and 2-9, 9-8, 10-79, 10-186
  narrow increment and 2-9, 9-8, 10-79, 10-186
  wide decrement and 2-9, 10-87, 10-192
  wide increment and 2-9, 10-88, 10-192
Skip instruction,
  accumulator 5-2
  device flags for 7-5
  executing 5-1
Skip instructions 2-7, 2-14, 2-22f, 3-8, 5-2f
Skip instructions, fixed-point logical 2-14
Skip on accumulator bit instructions, wide 2-23
Skip on accumulator bit set to one, wide 2-8, 10-168
Skip on accumulator bit set to zero, wide 2-8, 10-168
Skip on all bits set in accumulator,
  narrow 2-8, 10-98
  wide 2-8, 10-163
Skip on all bits set in memory,
  narrow 2-8
  wide 2-8
Skip on any bit set in accumulator,
  narrow 2-8, 10-99
  wide 2-8, 10-164
Skip on any bit set in memory,
  narrow 2-8
  wide 2-8
Skip on bit instruction, wide 2-14
Skip on CARRY (see ADC, ADD, AND, COM, INC, MOV, NEG,
  or SUB instruction)
Skip on condition instructions,
  floating-point 3-9
  logical 2-14
Skip on condition,
  fixed-point 2-7f
  floating-point 3-9
  testing BUSY or DONE flag and 7-3
Skip on greater than or equal to zero (N = 0) 3-9, 10-55
Skip on greater than zero (N and Z = 0) 3-9, 10-55
Skip on less than or equal to zero (N and Z = 1) 3-9, 10-55
Skip on less than zero (N = 1) 3-9, 10-56
Skip on no error (ANY = 0) 3-9, 10-57
Skip on no invalid input argument (INV = 0) 10-57
Skip on no mantissa overflow (MOF = 0) 3-9, 10-57
Skip on no overflow (OVF = 0) 3-9, 10-57
Skip on no overflow and no invalid input argument (OVF and INV = 0) 10-58
Skip on no overflow and no zero divide (OVF and INV = 0) 3-9
Skip on no underflow (UNF = 0) 3-9, 10-58
Skip on no underflow and no invalid input argument (UNF and INV = 0) 10-58
Skip on no underflow and no overflow (UNF and OVF = 0) 3-9, 10-58
Skip on no underflow and no zero divide (UNF and INV = 0) 3-9
Skip on no zero divide (INV = 0) 3-9
Skip on nonzero (Z = 0) 3-9, 10-57
Skip on nonzero bit 9-6ff, 10-110
Skip on nonzero bit, wide 2-8, 2-14, 9-8, 10-169
Skip on OVR flag reset 2-8, 10-111
Skip on valid byte 2-22
Skip on valid byte pointer 10-117
Skip on valid word address 2-22
Skip on valid word pointer 10-118
Skip on zero (Z = 1) 3-9, 9-6ff, 10-55, 10-116
Skip on zero bit 9-8
Skip on zero bit and set bit to one 9-6, 9-8, 10-117
Skip on zero bit and set bit to one, wide 2-8, 2-14, 9-8, 10-174

Skip on zero bit and set to one 9-8
Skip on zero bit and set to one, bit 9-8
Skip on zero bit, wide 2-8, 2-14, 9-8, 10-174
Skip option 2-7
Skip,
  add complement with optional 2-4, 2-6, 2-8, 10-2f
  add with optional 2-4, 2-6, 2-8, 10-3ff
  always 5-3
  AND with optional 2-6, 2-12ff, 10-6ff
  compare to limits and 9-8, 10-12
  fixed-point decrement word and 2-9
  fixed-point increment word and 2-9
  increment with optional 2-4, 2-6, 2-8f
  I/O 7-3, 10-110
  load physical and conditional 10-82
  move and 2-3
  move with optional 2-6, 2-8, 10-89f
  narrow backward search queue and 5-3, 6-5, 10-93f
  narrow forward search queue and 5-3, 6-5, 10-96f
  negate with optional 2-6, 2-13f
  never 5-3
  no 5-3, 10-51
  one's complement with optional 2-6, 2-14
  PC relative 5-3
  subtract with optional 2-4, 2-6, 2-8, 10-115f
  wide backward search queue and 5-3, 6-5, 10-122f
  wide compare to limits and 2-8, 9-8, 10-124
  wide forward search queue and 5-3f, 6-5, 10-150
Skipping a word 2-7
Skipping an instruction 3-8
Skipping, word 2-7
SKP 7-3, 10-110
SMRF 8-12, 10-110
SNB 9-6ff, 10-110
SNOVR 2-8, 10-111
Source descriptor for WDMOV or WDCMP 5-23
Source indicator 2-21
Source indicator, original 5-23f
Source pointer for WDMOV or WDCMP 5-23
Source segment 5-9ff
Source stack, copying arguments from the 5-12
Source, AND with complemented 10-6
Space,
  C/350 address 9-4ff
  logical address 8-1, 8-11
  memory address 1-1
  physical memory 8-11
  valid address 2-22
Special save and reset overflow mask, wide 4-5, 5-5, 9-3, 9-10f, 10-170f
Special save and set overflow mask, wide 4-5, 5-5, 9-3, 9-10f, 10-171
Special save and set OVK to one, wide 2-9
Special save and set OVK to zero, wide 2-9
Special save instruction, wide 5-5ff, 9-3f
Special save/reset overflow mask, wide 4-8
Special save/set overflow mask, wide 4-8
Specific instructions, interrupting 2-11
Specifying eight segment base registers 8-7
SPSR 2-9, 10-111
SPTE 8-5, 10-111
Square root double, floating-point 10-149
Square root evaluation, floating-point 3-10f
Square root instruction error, floating-point 3-14
Square root single, floating-point 10-149
Square root, floating-point 3-11
STA 9-8, 10-111
Stack 4-1
Stack access instructions, double-word 4-5
Stack base,
  initializing wide 4-2, 8-11
  loading wide 5-11, 5-13
  wide 4-2f, 4-7, 5-11, 5-13, 5-25, 7-9, 7-11, 8-11
Stack data instructions, wide 4-4
Stack definition, wide 5-15
Stack example operation, wide 5-7f
Stack fault 4-1, 5-12, 5-14, 5-16
Stack fault address parameter, vector 7-9
Stack fault code, wide 5-26, D-1f
Stack fault handler,
  first instruction of vector 7-13
  jumping to narrow 5-28
  narrow 5-27f
  saving pointer to wide 7-9
  vector 7-13
  wide 5-25
  writing narrow 5-28
  writing wide 5-26
Stack fault operation 5-13
Stack fault operations,
  narrow 5-26
  wide 5-25
Stack fault pointer, initializing wide 7-9
Stack fault recognition, enabling 5-25
Stack fault return block 4-8, 5-26, 5-28
Stack fault return block,
  narrow 5-28
  wide 5-26
Stack fault,
  disabling 4-7, 5-25, 5-27
  narrow 5-25
  servicing narrow 5-25, 5-27
  servicing wide 5-25
  wide 5-25
  words required beyond WSL for 4-8

Stack faults,
type of 5-25
wide 4-7
Stack for segment zero, redefining wide 8-11
Stack for temporary storage, wide 5-21
Stack instructions,
C/350 9-11
multi-word wide 4-8
Stack limit parameter, vector 7-9
Stack limit,
bit zero of narrow 5-27
bit zero of wide 5-26
initializing wide 8-11
loading wide 5-11, 5-13
lower 4-2
narrow 5-26ff, 9-2
setting bit zero of narrow 5-27f
setting narrow 5-27
upper 4-2
wide 4-3, 4-7, 5-11, 5-13, 5-25, 7-9, 7-11, 8-11
zero-extending vector 7-9
Stack location, defining 4-2
Stack management instructions, C/350 9-11
Stack management registers, defined wide 1-4
Stack management,
narrow 1-4
wide 1-4
Stack operation 1-16
Stack operation example, wide 5-7
Stack operation,
overflow 5-26, D-2
underflow 5-26
Stack operations 4-1
Stack operations,
managing narrow 4-1
managing wide 4-1
wide 4-1
Stack overflow 4-6f
Stack overflow condition 4-3, 5-12
Stack overflow detection, enabling vector 7-9
Stack overflow fault,
checking for 4-3
checking for wide 4-3
detecting wide 5-25
disabling narrow 5-27
disabling wide 4-7, 5-25
narrow 5-26f
Stack overflow,
checking for 5-14
checking for vector 7-13
destination 5-12
Stack parameters,
managing 4-1
narrow 9-2
pushing previously saved 7-9
saving wide 7-9
vector 7-9
wide 4-1
Stack pointer contents, store accumulator in 10-113
Stack pointer parameter, vector 7-9
Stack pointer wraparound, narrow 5-27
Stack pointer,
bit zero of narrow 5-27
bit zero of wide 5-26
comparing wide 4-3
decrementing wide 4-3f
incrementing wide 4-3f
initializing wide 4-3, 8-11
loading wide 5-11, 5-13
modify 9-11, 10-91
narrow 5-26ff, 9-2
saving wide 7-9, 7-11
setting bit zero of narrow 5-27f
setting narrow 5-27
setting wide 4-3
storing wide 5-11, 5-13, 8-11
updating wide 5-25
wide 4-2f, 4-7, 5-12, 5-15, 5-25, 7-9
wide modify 4-4, 5-26, 9-11, 10-158
zero-extending vector 7-9
Stack register contents, modifying 4-2
Stack register initialization 5-15
Stack register instructions, wide 4-4
Stack register, initializing wide 4-2, 4-4
Stack registers 4-1f
Stack registers,
initializing four wide 7-9, 7-11
saving wide 7-9
wide 4-2
Stack return block instructions, wide 4-5
Stack to lower 128 Kbyte, restricting vector 7-9
Stack underflow 4-7, 5-25, D-2
Stack underflow condition 4-3
Stack underflow detection, enabling vector 7-9
Stack underflow fault,
detecting wide 5-25
disabling narrow 5-27
disabling wide 4-7, 5-25
narrow 5-26
Stack underflow,
enabling narrow 9-2
narrow 5-27
wide 4-3
Stack,
accessing double word in a 4-4
C/350 9-2
constructing double word for vector 7-12
copying arguments from the source 5-12
decrementing word in 2-21
defined 1-4
defined narrow 5-25

defining limits of narrow 9-2
initializing narrow 5-27
initializing wide 4-3, 4-6
loading the vector 7-9
location of narrow 5-27
lower limit of the wide 4-2
narrow 1-4, 4-1, 5-20, 9-2, 9-4
overwriting data in area beyond the 5-25
popping narrow 5-26
pushing arguments onto a wide 5-7
pushing double word onto vector 7-12
pushing narrow 5-26
pushing wide return block onto vector 7-9, 7-11
redefining wide 5-10f, 5-13
retrieving data from the wide 4-3
store in 2-21, 10-33
storing data in the 4-2
top location of wide 4-3
upper limit of the wide 4-3
vector 7-7, 7-9, 7-12
wide 1-4, 4-1, 5-4, 5-6f, 5-12f, 5-20
Stack-referenced instructions, C/350 9-1
STAFP 4-4, 10-112
Standard wide return block 4-6
Starting a device 7-5
Starting address of block to transfer 7-2
Starting address of sub-block to transfer 7-2
STASB 4-4, 10-112
STASL 4-4, 10-112
STASP 4-4, 10-112
State instruction, load floating-point 3-14
State of processor,
restoring 8-11
saving current 8-8, 8-11
State,
device 7-5
internal processor 7-9
pop floating-point 3-12, 9-9, 10-51f
push floating-point 3-12, 9-9, 10-52f
STATS 2-3, 4-5, 10-113
Status and accumulators,
pop floating-point 10-51f
push floating-point 10-52f
Status field in context block 8-11
Status flag, error 3-12
Status flags, testing 3-8
Status instructions, floating-point 3-12
Status of pages 8-5
Status register format, processor 2-10
Status register from AC0, store processor 10-111
Status register in AC0, load processor 10-83
Status register,
floating-point 1-3, 3-4, 3-6, 3-12f
load floating-point 9-9, 10-48
processor 1-2, 2-9, 5-13, 5-19f, 5-23, 5-26, 7-6, 9-1, 9-4
setting processor 5-19, 5-21, 7-3
store floating-point 9-9, 10-59, 10-75
testing the floating-point 3-12
Status,
floating-point 3-11ff
load floating-point 10-74
mantissa 3-12f
storing floating-point 5-21
testing machine 5-2
STB 9-8, 10-113
STI 9-9, 10-113f
STIX 2-24, 5-21f, 9-9, 10-114, D-2
STIX, destination indicator for 5-23
Storage capacity, exceeding processor 1-2
Storage, wide stack for temporary 5-21
Store accumulator 9-8, 10-111
Store accumulator in double word addressed by WSP 2-3, 4-5
Store accumulator in stack pointer contents 10-113
Store accumulator in WFP 4-4, 10-112
Store accumulator in WSB 4-4, 10-112
Store accumulator in WSL 4-4, 10-112
Store accumulator in WSP 4-4, 10-112
Store accumulator,
extended 9-8, 10-41
narrow 2-3, 9-8, 10-80, 10-187
wide 2-3, 10-89, 10-193
Store byte 2-20, 9-8, 10-86, 10-113, 10-190
Store byte,
extended 9-8, 10-41
wide 2-20, 9-8, 10-172
Store floating-point double 9-9, 10-59, 10-76, 10-183
Store floating-point single 9-9, 10-60, 10-76, 10-183
Store floating-point status register 9-9, 10-59, 10-75
Store if equal, wide mask skip and 6-5, 10-157
Store in stack 2-21, 10-33
Store integer 9-9, 10-113f
Store integer extended 9-9, 10-114
Store integer extended, wide 9-9, 10-173
Store integer, wide 10-172
Store modified and referenced bits 8-12, 10-110
Store page table entry 8-5, 10-111
Store processor status register from AC0 10-111
Store PSR from AC0 2-9
Storing accumulators 5-7
Storing byte address in accumulator 2-23
Storing carry 5-7
Storing data in the stack 4-2
Storing decimal number 2-20
Storing device dependent constants or variables 7-12
Storing floating-point data 3-4
Storing floating-point status 5-21

Storing FPSR 3-12
Storing processor parameters 8-12
Storing PSR 5-7
Storing return address 5-7, 7-7
Storing single precision 3-4
Storing the floating-point result 3-6
Storing transient overflow condition 2-10
Storing wide frame pointer 5-7, 5-11, 5-13, 8-11
Storing wide stack pointer 5-11, 5-13, 8-11
String arithmetic instructions, decimal 2-22
String of bytes,
  manipulating 2-21
  moving 2-21
  searches a 2-23
String of characters,
  converting 2-20
  inserting 2-20
String,
  packed decimal 2-17
  unpacked decimal 2-17
Structure, implicit data 2-15
SUB 2-4, 2-6, 2-8, 10-115f
Sub-block to transfer, starting address of 7-2
Subopcode causing fault, byte pointer to 5-23
Subopcode, byte pointer to fault 5-24
Subprogram instructions, EDIT 2-21
Subprogram,
  edit 5-4
  end edit 2-21
  WEDIT 5-4
Subroutine 9-4
Subroutine call 2-9, 3-13, 4-1, 4-3, 4-5, 4-8, 5-4ff, 5-8, 5-13
Subroutine call,
  aborting 5-11f
  executing 5-1
  illegal outward 5-9
  returning from 3-13, 4-1, 4-3, 4-5
Subroutine from a 32-bit subroutine, calling C/350 9-4
Subroutine handler 2-10
Subroutine instructions, sequence of 5-6
Subroutine return 5-4, 5-13
Subroutine,
  address first instruction of 5-10, 5-12
  call 10-66f, 10-177f
  calling C/350 subroutine from a 32-bit 9-4
  changing C/350 9-4
  executing 5-12
  exiting 4-3
  expanding an ECLIPSE C/350 9-4
  extended jump to 9-10, 10-37
  jump to 5-4ff, 9-3f, 9-10, 10-65, 10-76, 10-183
  jumping to 5-7
  last instruction of 5-13
  passing arguments to a 5-7
  push and jump to 4-5, 4-8, 5-6
  returning from 3-13, 4-1, 4-3
Subtract double (FPAC from FPAC) 3-7, 10-55
Subtract double (memory from FPAC) 3-7, 9-9, 10-56, 10-75, 10-182
Subtract immediate 2-4, 10-108
Subtract immediate,
  narrow 10-80, 10-100, 10-187
  wide 10-88, 10-166, 10-193
Subtract instructions, decimal fixed-point 2-22
Subtract memory word 2-4
Subtract memory word,
  narrow 10-81, 10-187
  wide 10-89, 10-194
Subtract single (FPAC from FPAC) 3-7, 10-59
Subtract single (memory from FPAC) 3-7, 9-9, 10-56, 10-75, 10-182
Subtract with optional skip 2-4, 2-6, 2-8, 10-115f
Subtract,
  decimal 10-31f
  narrow 2-4, 10-100
  wide 2-4, 10-173
Subtraction instructions, fixed-point 2-4
Subtraction, floating-point 3-5, 3-7
Supporting 16-bit programs 4-1
Supporting 32-bit programs 4-1
Supporting C/350 program development 9-2
Supporting devices 7-1
Suppress, insert character 10-25
Suppression, move numeric with zero 2-21, 10-30
Swapping memory pages 8-8
Swapping two bytes 2-6
System calls, I/O 7-5
System identification instructions 8-12
System management 1-6, 8-1
System protection capabilities 1-1
System reset 2-10
System,
  device interrupt 7-5
  interrupt 1-5, 7-5, 8-11
  invoking interrupt 7-5, 8-11
  kernel operating 1-6
SZB 9-6ff, 10-116
SZBO 9-6ff, 10-117

T

T to one, set 10-34
T to zero, set 10-34
T, add to P depending on 10-21
Table depth fault, page 8-7
Table entries, page 8-1
Table entry,
  load page 8-5, 10-83
  page 8-7f
  two-level page table 8-7f

store page 8-5, 10-111
vector 7-12
Table frame (frame number), root page 8-3
Table frame, root page 8-2
Table in memory, page 8-1
Table logical word address,
one-level page 8-7
two-level page 8-7
Table of referenced and modified bits 8-1
Table offset, page 8-3
Table page, addressing another page 8-7
Table translation,
example of one-level page 8-9
example of two-level page 8-10
one-level page 8-8
two-level page 8-7f
Table validity protection fault, page 8-8
Table validity violation, page 5-15
Table,
accessing page 8-5
address of first entry in page 8-2ff
base of vector 7-12
device control 7-12f
one-level page 8-2f, 8-7ff
page E-1
powers of 2 C-1
two-level page 8-2f
vector 7-11f
word zero of vector 7-11
Tables, page 8-1, 8-5
Tail, enqueue towards the 10-40
Tangent double, floating-point 10-151
Tangent instruction error, floating-point 3-14
Tangent single, floating-point 10-151
Tangent, floating-point 3-11
TE fault mask 5-20
TE flag 4-8
TE mask,
resetting 3-12
setting 3-12f, 5-21
Temporary storage, wide stack for 5-21
Termination offset 5-3
TES flag, defined 2-11
Testing a device flag 7-3f
Testing BUSY flag and skip on condition 7-3
Testing DONE flag and skip on condition 7-3
Testing for a condition 2-23
Testing for identical segments 5-12
Testing interrupt level word 7-9
Testing machine status 5-2
Testing result for a condition 3-8
Testing status flags 3-8
Testing the divisor for zero 3-8
Testing the floating-point status register 3-12
Testing the INV flag 3-9
Testing the results of an operation 2-7
Three of SBR, setting bit 7-2
Times, insert character j 10-25
Top location of wide stack 4-3
Towards the head, enqueue 10-39f
Towards the tail, enqueue 10-40
Transfer instructions, segment 5-8
Transfer rates, data 7-1
Transfer to another segment, initiating a 5-8
Transfer,
address of first word for I/O 7-2
addressing memory for I/O data 7-2
direction of 5-8, 7-2
direction of I/O data 7-2
initializing burst multiplexor channel 7-2
initializing data channel 7-2
no I/O 7-3, 10-97
parameters of I/O data 7-2
requesting I/O data 7-5
starting address of block to 7-2
Transferring blocks of words, between memory and device 1-6
Transferring bytes, between ac and device 1-5
Transferring call arguments 5-10ff
Transferring data 1-5
Transferring formatted data 2-20
Transferring I/O data 7-2
Transferring page to disk 8-11
Transferring program control 1-8, 4-2, 4-4, 5-6, 5-10, 5-21, 7-13
Transferring program control to another segment 5-8ff
Transferring to device interrupt routine 7-13
Transferring words,
between ac and device 1-5
between memory and device 1-6
Transfers between device and accumulator 7-1
Transfers between device and memory 7-1
Transfers between high speed device and memory 7-1
Transfers between medium speed device and memory 7-1
Transfers between memory and device 7-1ff
Transfers, checking data 5-12
Transient overflow condition, storing 2-10
Translate and compare,
character 9-8, 10-18f
wide character 2-20, 2-23, 9-8, 10-129f
Translating logical address 1-7, 7-2f, 8-2, 8-5
Translating logical address, most significant bits 7-2
Translation level flag 8-3
Translation,
device address 7-1, 7-6
example of one-level page table 8-9
example of two-level page table 8-10
memory address 8-2f, 8-7
one-level page table 8-8

Translator, purging address 8-12, 10-101
Trap disable,
  fixed-point 2-9, 10-61
  floating-point 3-12, 10-60
Trap enable mask 3-12f, 5-19
Trap enable,
  fixed-point 2-9, 10-61
  floating-point 3-12, 10-60
Trap,
  defined 5-19f
  floating-point A-2
Trigger is one, opcode pointer if 2-21
Trigger to one or zero, set 2-21
Trigger, insert character depending on 2-21
Trigonometric functions,
  floating-point 3-10f
  floating-point inverse 3-10f
Trojan horse pointer, protecting against 5-12
Trojan horse pointers 5-12
True value of exponent 3-2
True zero 3-1, 3-6, 3-13, E-2
True, character move until 9-8, 10-13f
Truncating 3-4
Truncating intermediate mantissa 3-6ff
Truncating intermediate result 3-13
Truncating the mantissa 3-4
Truncating the result 3-5
Two bytes, swapping 2-6
Two floating-point numbers, compare (set N and Z) 3-8f, 10-43
Two Kbyte address boundary 8-3
Two of SBR, setting bit 7-1ff
Two operands, signs of the 3-6
Two's complement 2-2
Two's complement number,
  calculating a 2-8, 5-19
  out of range calculating a 5-20
Two-level page table 8-2f
Two-level page table logical word address 8-7
Two-level page table translation 8-7f
Two-level page table translation, example of 8-10
Type 6 or 7, invalid data 2-24, 5-22, D-2
Type field 2-16
Type of data 2-16
Type of page access 8-5
Type of segment access 8-2
Type of stack faults 5-25
Type,
  checking for valid data and 5-13, 5-21ff
  invalid sign code for data 2-24, 5-22
  return block 2-24, 5-22
Types of narrow fault return blocks 5-24
Types of wide fault return blocks 5-23

U

Underflow condition, stack 4-3
Underflow detection, enabling vector stack 7-9
Underflow fault,
  detecting floating-point 5-20
  detecting wide stack 5-25
  disabling narrow stack 5-27
  disabling wide stack 4-7, 5-25
  narrow stack 5-26
Underflow flag, exponent 3-12
Underflow stack operation 5-26
Underflow,
  enabling narrow stack 9-2
  exponent 3-6
  stack 4-7, 5-25, 5-27, D-2
  wide stack 4-3
UNF flag 3-12, 5-20
UNF flag of the FPSR 3-10
Unimplemented instruction fault code D-1
Unimplemented instruction protection violation 5-17
Unimplemented instruction violation 5-15
Unimplemented instructions 5-19
Unimplemented opcode 5-14ff
Unimplemented opcode violation 5-15
Unique device, identifying 7-4
Unmapped memory addressing 1-5
Unmapped mode 7-3
Unpacked byte 2-15
Unpacked decimal 2-15
Unpacked decimal string 2-17
Unsigned divide 2-5, 10-26
Unsigned multiply 2-5, 10-91
Unsigned skip if accumulator greater than immediate, wide 2-8, 10-174
Unsigned skip if accumulator less than or equal to immediate, wide 2-8, 10-175
Unsigned skip if ACS greater than ACD, wide 2-8, 10-175
Unsigned skip if ACS greater than or equal to ACD, wide 2-8, 10-175
Updating program counter 5-14, 5-25, 5-28
Updating wide stack pointer 5-25
Updating WSP reserved memory location 5-25
Upper limit of the wide stack 4-3
Upper stack limit 4-2
Upward program compatibility 4-1, 9-1
Using an accumulator 1-2
Using device map 7-2
Using the wide frame pointer 4-3

V

Valid address space 2-22
Valid addressing range 8-3
Valid ASCII data type 2-24
Valid byte pointer, skip on 10-117

Valid byte, skip on 2-22
Valid data and type, checking for 5-13, 5-21ff
Valid decimal data type 2-24
Valid gate 5-11
Valid operations, checking for 5-13
Valid page 8-5, 8-8
Valid page access 8-5
Valid read 8-8
Valid read access 8-6
Valid referenced page 8-8
Valid segment 8-3
Valid segment access 8-2
Valid segment check 8-3
Valid segment reference 8-7
Valid word pointer, skip on 2-22, 10-118
Valid write 8-8
Validate I/O access 8-2
Validation, access 8-8
Validity bit protection fault code D-1
Validity bit, checking 5-11
Validity flag,
  I/O 1-17, 7-2, 8-3
  segment 8-2
Validity protection fault,
  page table 8-8
  segment 8-2
Validity protection violation 5-17
Validity violation,
  page table 5-15
  segment 5-15
Value of exponent, true 3-2
Value of P, current 5-23f
Value, absolute 10-42
Variable to constant, comparing DO-loop 5-3
Variable, incrementing a DO-loop 5-3
Variables, storing device dependent 7-12
VBP 2-22, 5-12, 10-117
Vector interrupt instruction 7-7
Vector interrupt processing 7-7, 7-9
Vector interrupt service, concluding 7-11
Vector interrupt,
  I/O 2-9
  servicing 7-3
  servicing base level 7-9, 10-161
  servicing intermediate level 7-11ff, 10-160f
Vector on interrupting device 10-190
Vector on I/O interrupt 4-8
Vector stack 7-7, 7-9, 7-12
Vector stack fault address parameter 7-9
Vector stack fault handler 7-13
Vector stack limit parameter 7-9
Vector stack limit, zero-extending 7-9
Vector stack overflow detection, enabling 7-9
Vector stack overflow, checking for 7-13
Vector stack parameters 7-9
Vector stack pointer, zero-extending 7-9
Vector stack to lower 128 Kbyte, restricting 7-9
Vector stack underflow detection, enabling 7-9
Vector stack,
  constructing double word for 7-12
  loading the 7-9
  pushing double word onto 7-12
  pushing wide return block onto 7-9, 7-11
Vector table 7-11f
Vectored gate array 5-11
Vectored interrupt handler 7-7, 10-190
Violation fault codes D-1
Violation fault codes, protection D-1
Violation fault, protection 1-8f, 5-13
Violation sequence, protection 5-18
Violation,
  detecting protection 7-2
  execute access 5-15
  I/O instruction 5-15
  indirect addressing 1-11, 5-15
  inward reference 5-15
  page table validity 5-15
  privileged instruction 5-15
  protection 1-16, 8-2f, 8-5f
  read access 5-15
  segment crossing 5-15
  segment validity 5-15
  unimplemented instruction 5-15
  unimplemented opcode 5-15
  write access 5-15
Violations,
  multiple protection 5-14ff
  protection 5-14ff
Virtual memory 8-1
Virtual memory, defined 1-6
Voltage ranges, detecting proper power 7-5
VWP 2-22, 5-12, 10-118

W

WADC 2-4, 10-118
WADD 2-4, 10-119
WADDI 2-4, 10-119
WADI 2-4, 10-119
WANC 2-13, 10-119
WAND 2-13, 10-120
WANDI 2-13, 10-120
WASH 2-6, 10-120
WASHI 2-6, 10-121
WBLM 2-3, 9-8, 10-121f
WBR 5-2f, 9-10, 10-122
WBStc 5-3f, 6-5, 10-122f
WBTO 2-13, 9-8, 10-123f
WBTZ 2-13, 9-8, 10-124
WCLM 2-8, 9-8, 10-124
WCMP 2-23, 9-8, 10-125f
WCMT 2-20, 5-12, 9-8, 10-126f

WCMV 2-20, 5-12, 9-8, 10-127f
WCOB 2-13, 9-8, 10-128
WCOM 2-13, 10-128
WCST 2-23f, 10-128f
WCTR 2-20, 2-23f, 9-8, 10-129f
WDCMP 2-22, 2-24, 5-22, 10-130f, D-2
WDCMP,
  source descriptor for WDMOV or 5-23
  source pointer for WDMOV or 5-23
WDCMP, WDINC, or WDDEC, destination pointer for
    WDMOV, 5-23
WDDEC 2-22, 2-24, 5-22, 10-131, D-2
WDDEC, destination pointer for WDMOV, WDCMP, WDINC,
    or 5-23
WDINC 2-22, 2-24, 5-22, 10-131, D-2
WDINC, or WDDEC, destination pointer for WDMOV, WDCMP, 5-23
WDIV 2-5, 10-132
WDIVS 2-5, 10-132
WDMOV 2-20, 2-24, 5-22, 10-132f, D-2
WDMOV or WDCMP,
  source descriptor for 5-23
  source pointer for 5-23
WDMOV, WDCMP, WDINC, or WDDEC, destination pointer
    for 5-23
WDPOP 2-9, 8-11, 10-133
WEDIT 2-20ff, 4-8, 5-4f, 5-21f, 9-8, 10-133ff, D-2
WEDIT fault instruction, address of 5-23
WEDIT subprogram 5-4
WFACOSD 3-11, 3-14, 10-135
WFACOSS 3-11, 3-14, 10-135f
WFASIND 3-11, 3-14, 10-136
WFASINS 3-11, 3-14, 10-136f
WFATAND 3-11, 10-137
WFATANS 3-11, 10-137
WFATN2D 3-11, 10-138
WFATN2S 3-11, 3-14, 10-138
WFCOSD 3-11, 10-139
WFCOSS 3-11, 10-139
WFEXPD 3-11, 3-14, 10-140
WFEXPS 3-11, 3-14, 10-140
WFFAD 3-3f, 3-13, 9-9, 10-141
WFLAD 3-3f, 9-9, 10-141
WFLG2D 3-11, 3-14, 10-141f
WFLG2S 3-11, 3-14, 10-142
WFLNGD 3-11, 3-14, 10-142
WFLNGS 3-11, 3-14, 10-143
WFLOGD 3-11, 3-14, 10-143
WFLOGS 3-11, 3-14, 10-144
WFP 4-3, 4-6
WFP,
  load accumulator with 4-4, 10-67
  store accumulator in 4-4, 10-112
WFPOP 3-4, 3-12, 4-5, 4-8, 9-9, 10-144f
WFPSH 3-4, 3-12, 4-5, 4-8, 9-9, 10-146
WFPWRD 3-11, 3-14, 10-147
WFPWRS 3-11, 3-14, 10-147
WFSIND 3-11, 10-148
WFSINS 3-11, 10-148
WFSQRD 3-11, 3-14, 10-149
WFSQRS 3-11, 3-14, 10-149
WFStc 5-3f, 6-5, 10-150
WFTAN2D 3-14
WFTAND 3-11, 3-14, 10-151
WFTANS 3-11, 3-14, 10-151
WHLV 2-5, 10-152
Wide add 2-4, 10-119
Wide add complement 2-4, 10-118
Wide add immediate 2-4, 10-86, 10-119, 10-191
Wide add memory word to accumulator 2-4, 10-86, 10-191
Wide add with narrow immediate 2-4, 10-159
Wide add with wide immediate 2-4, 10-119
Wide AND 2-13, 10-120
Wide AND immediate 2-13, 10-120
Wide AND, with complemented source 2-13
Wide arithmetic shift 10-120
Wide arithmetic shift instructions 2-6
Wide arithmetic shift with narrow immediate 10-121
Wide backward search queue and skip 5-3, 6-5, 10-122f
Wide bit addressing format 1-15
Wide block move 2-3, 9-8, 10-121f
Wide branch 9-10, 10-122
Wide character compare 2-23, 9-8, 10-125f
Wide character move 2-20, 9-8, 10-127f
Wide character move until true 2-20, 9-8, 10-126f
Wide character scan until true 2-23, 10-128f
Wide character translate and compare 2-20, 2-23, 9-8, 10-129f
Wide compare to limits and skip 2-8, 9-8, 10-124
Wide complement 10-128
Wide complement (one's complement) 2-13
Wide count bits 2-13, 9-8, 10-128
Wide data, converting to 2-2
Wide decimal compare 2-22, 10-130f
Wide decimal decrement 2-22, 10-131
Wide decimal increment 2-22, 10-131
Wide decimal move 2-20, 10-132f
Wide decrement and skip if zero 2-9, 10-87, 10-192
Wide divide 2-5, 10-132
Wide divide memory word 2-5, 10-87, 10-191
Wide DO until greater than 10-87, 10-192
Wide edit 4-8, 9-8, 10-133ff
Wide edit of alphanumeric 5-5
Wide exchange 2-3, 10-176
Wide exclusive OR 2-13, 10-177
Wide exclusive OR immediate 2-13, 10-177
Wide extended operation 5-5, 9-10f, 10-176

Wide fault return blocks, types of 5-23
Wide fix from floating-point accumulator 3-3f, 3-13, 9-9, 10-141
Wide float from fixed-point accumulator 3-3f, 9-9, 10-141
Wide floating-point fault 5-20f
Wide floating-point pop 3-4, 4-5, 4-8, 10-144f
Wide floating-point push 3-4, 4-5, 4-8, 10-146
Wide forward search queue and skip 5-3f, 6-5, 10-150
Wide frame pointer 4-2f, 5-6ff, 5-15, 7-9
Wide frame pointer,
  initializing 4-3, 5-11
  loading 5-12f
  saving 7-9
  storing 5-7, 5-11, 5-13, 8-11
  using the 4-3
Wide halve (AC/2) 2-5, 10-152
Wide immediate,
  wide add with 10-119
  wide load with 10-153
Wide inclusive OR 2-13, 10-152
Wide inclusive OR immediate 2-12f, 10-152
Wide increment (no skip) 2-4, 10-152
Wide increment and skip if zero 2-9, 10-88, 10-192
Wide instruction opcodes A-1
Wide instructions E-2
Wide load accumulator 2-3, 10-88, 10-193
Wide load byte 2-20, 9-8, 10-153
Wide load integer 9-9, 10-153
Wide load integer extended 9-9, 10-154
Wide load map 10-154
Wide load sign 2-13, 2-23, 9-8, 10-156f
Wide load with wide immediate 2-3, 10-153
Wide locate and reset lead bit 2-13, 10-155
Wide locate lead bit 2-13, 10-155
Wide logical shift 2-14, 10-155
Wide logical shift immediate 2-14, 10-156
Wide logical shift left immediate 2-14
Wide logical shift with narrow immediate 10-156
Wide mask skip and store if equal 6-5, 10-157
Wide modify stack pointer 4-4, 5-26, 9-11, 10-158
Wide move 2-3, 10-158
Wide move right 2-21, 10-158
Wide multiply 2-5, 10-159
Wide multiply memory word 2-5, 10-88, 10-193
Wide negate 2-13, 10-159
Wide pop block 2-9, 4-5, 4-8, 5-5f, 5-8, 5-14, 5-26, 9-10f, 10-160f
Wide pop block and execute 2-9, 4-5, 4-8, 5-8, 5-14, 5-26, 9-10f, 10-160f
Wide pop context block 2-9, 8-8, 8-11f, 10-133
Wide pop multiple accumulators 2-3, 4-1, 4-5, 4-8, 9-8, 9-11, 10-160
Wide pop operation 5-25
Wide pop PC and jump 4-5, 4-8, 5-5, 9-10f, 10-161
Wide push accumulators 2-3, 4-5, 4-8, 9-8, 10-161
Wide restore 2-9, 4-8, 9-10f, 10-162
Wide restore from an I/O interrupt 4-5, 5-8, 7-7
Wide return 2-9, 4-8, 5-5, 5-8, 9-3f, 9-10f, 10-163
Wide return block 4-3, 5-7, 5-25
Wide return block for decimal data, type 1 fault 5-23
Wide return block onto vector stack, pushing 7-9, 7-11
Wide return block,
  popping 4-3, 5-4, 7-7
  pushing 5-4, 5-6, 5-12, 5-19ff
  standard 4-6
Wide return instruction 5-6f, 5-13, 9-3f
Wide return via wide save 4-5
Wide save and reset overflow mask 4-5, 5-5, 10-164f
Wide save and set overflow mask 4-5, 5-5, 10-165
Wide save and set OVK to one 2-9f
Wide save and set OVK to zero 2-9
Wide save instruction 4-3, 5-12
Wide save, wide return via 4-5
Wide save/reset OVK mask 4-8
Wide save/set OVK mask 4-8
Wide set bit to one 2-13, 9-8, 10-123f
Wide set bit to zero 2-13, 9-8, 10-124
Wide signed divide 2-5, 10-132
Wide signed multiply 2-5, 10-159
Wide signed skip if ACS greater than ACD 2-8, 10-167
Wide signed skip if ACS greater than or equal to ACD 2-8, 10-167
Wide signed skip if ACS less than ACD 2-8
Wide signed skip if ACS less than or equal to ACD 2-8, 10-168
Wide signed skip if less than 10-169
Wide skip if accumulator equal to immediate 2-23, 10-166
Wide skip if accumulator greater than immediate 2-8, 10-167
Wide skip if accumulator less than or equal to immediate 2-8, 10-169
Wide skip if accumulator not equal to immediate 2-8, 10-170
Wide skip if ACS equal to ACD 2-8, 10-166
Wide skip if ACS not equal to ACD 2-8, 10-170
Wide skip if equal to 2-23
Wide skip if equal to immediate 2-8
Wide skip on accumulator bit instructions 2-23
Wide skip on accumulator bit set to one 2-8, 10-168
Wide skip on accumulator bit set to zero 2-8, 10-168
Wide skip on all bits set in accumulator 2-8, 10-163
Wide skip on all bits set in memory 2-8, 10-163
Wide skip on any bit set in accumulator 2-8, 10-164
Wide skip on any bit set in memory 2-8, 10-164
Wide skip on bit instruction 2-14
Wide skip on nonzero bit 2-8, 2-14, 9-8, 10-169
Wide skip on zero bit 2-8, 2-14, 9-8, 10-174
Wide skip on zero bit and set bit to one 2-8, 2-14, 9-8, 10-174

Wide special save and reset overflow mask 4-5, 5-5, 9-3, 9-10f, 10-170f
Wide special save and set overflow mask 4-5, 5-5, 9-3, 9-10f, 10-171
Wide special save and set OVK to one 2-9
Wide special save and set OVK to zero 2-9
Wide special save instruction 5-5ff, 9-3f
Wide special save/reset OVK mask 4-8
Wide special save/set OVK mask 4-8
Wide stack 1-4, 4-1, 5-4, 5-6f, 5-12f, 5-20
Wide stack base 4-2f, 4-7, 5-11, 5-13, 5-25, 7-9, 7-11, 8-11
Wide stack base,
  initializing 4-2, 8-11
  loading 5-11, 5-13
Wide stack data instructions 4-4
Wide stack definition 5-15
Wide stack example operation 5-7f
Wide stack fault 5-25
Wide stack fault code 5-26, D-1f
Wide stack fault handler 5-25
Wide stack fault handler,
  saving pointer to 7-9
  writing 5-26
Wide stack fault operations 5-25
Wide stack fault pointer, initializing 7-9
Wide stack fault return block 5-26
Wide stack fault, servicing 5-25
Wide stack faults 4-7
Wide stack for segment zero, redefining 8-11
Wide stack for temporary storage 5-21
Wide stack instructions, multi-word 4-8
Wide stack limit 4-3, 4-7, 5-11, 5-13, 5-25, 7-9, 7-11, 8-11
Wide stack limit,
  bit zero of 5-26
  initializing 8-11
  loading 5-11, 5-13
Wide stack management 1-4
Wide stack operation example 5-7
Wide stack operations 4-1
Wide stack overflow fault,
  checking for 4-3
  detecting 5-25
  disabling 4-7, 5-25
Wide stack parameters 4-1
Wide stack parameters, saving 7-9
Wide stack pointer 4-2f, 4-7, 5-12, 5-15, 5-25, 7-9
Wide stack pointer,
  bit zero of 5-26
  comparing 4-3
  decrementing 4-3f
  incrementing 4-3f
  initializing 4-3, 8-11
  loading 5-11, 5-13
  saving 7-9, 7-11
  setting 4-3
  storing 5-11, 5-13, 8-11
  updating 5-25
Wide stack register instructions 4-4
Wide stack register, initializing 4-2, 4-4
Wide stack registers 4-2
Wide stack registers,
  initializing 4-4, 7-9, 7-11
  saving 7-9
Wide stack return block instructions 4-5
Wide stack underflow 4-3
Wide stack underflow fault,
  detecting 5-25
  disabling 4-7, 5-25
Wide stack,
  initializing 4-3, 4-6
  lower limit of the 4-2
  pushing arguments onto a 5-7
  redefining 5-10f, 5-13
  retrieving data from the 4-3
  top location of 4-3
  upper limit of the 4-3
Wide store accumulator 2-3, 10-89, 10-193
Wide store byte 2-20, 9-8, 10-172
Wide store integer 10-172
Wide store integer extended 9-9, 10-173
Wide subtract 2-4, 10-173
Wide subtract immediate 10-88, 10-166, 10-193
Wide subtract memory word 10-89, 10-194
Wide unsigned skip if accumulator greater than immediate 2-8, 10-174
Wide unsigned skip if accumulator less than or equal to immediate 2-8, 10-175
Wide unsigned skip if ACS greater than ACD 2-8, 10-175
Wide unsigned skip if ACS greater than or equal to ACD 2-8, 10-175
WINC 2-4, 10-152
WIOR 2-12f, 10-152
WIORI 2-12f, 10-152
WLDAI 2-3, 10-153
WLDB 2-20, 9-8, 10-153
WLDI 2-20, 2-24, 3-3, 5-22, 9-9, 10-153, D-2
WLDIX 2-20, 2-24, 3-3, 5-22, 9-9, 10-154, D-2
WLMP 7-3, 10-154
WLOB 2-13, 10-155
WLRB 2-13, 10-155
WLSH 2-14, 10-155
WLSHI 2-14, 10-156
WLSI 2-14, 10-156
WLSN 2-13, 2-23f, 5-22, 9-8, 10-156f, D-2
WMESS 6-5, 10-157
WMOV 2-3, 10-158
WMOVR 2-21, 10-158
WMSP 4-4, 5-26, 9-11, 10-158, D-2

WMUL 2-5, 10-159
WMULS 2-5, 10-159
WNADI 2-4, 10-159
WNEG 2-13, 10-159
Word address field 1-14f
Word address,
  one-level page table logical 8-7
  pushing 4-5
  skip on valid 2-22
  two-level page table logical 8-7
Word addressed by WSP, store accumulator in double 4-5
Word addressing,
  format C/350 9-5
  formats 1-9
Word and skip,
  fixed-point decrement 2-9
  fixed-point increment 2-9
Word block in a stack, accessing double 4-4
Word boundary 1-2, 3-2
Word for I/O transfer, address of first 7-2
Word for vector stack, constructing double 7-12
Word in a stack, accessing double 4-4
Word in device control table, PSR 7-13
Word in stack, decrementing 2-21
Word indirect pointer, single 9-7
Word of interrupt handler, first 7-7
Word offset 1-15, 9-6
Word onto vector stack, pushing double 7-12
Word operand 1-2, 1-13
Word operand, double 1-2, 1-13
Word pointer 1-15, 5-12, 9-6
Word pointer,
  converting byte pointer to 2-21
  single 9-5f
  skip on valid 10-118
Word to accumulator,
  narrow add memory 10-78, 10-184
  wide add memory 10-86, 10-191
Word zero of vector table 7-11
Word,
  decrementing interrupt level 7-9
  incrementing interrupt level 7-9
  load accumulator with double 10-68
  loading current interrupt mask 7-12
  narrow divide memory 10-78, 10-185
  narrow multiply memory 10-80, 10-187
  narrow subtract memory 10-81, 10-187
  pushing a double 4-3
  read high 10-54
  reading high floating-point 3-3
  testing interrupt level 7-9
  wide divide memory 10-87, 10-191
  wide multiply memory 10-88, 10-193
  wide subtract memory 10-89, 10-194
Words required beyond WSL for stack fault 4-8
Words required beyond WSL, number of 4-8
Words, reserved memory 5-25
WPOP 2-3, 4-1, 4-5, 4-8, 9-8, 9-11, 10-160
WPOP, popping with 4-5
WPOPB 2-9, 4-5, 4-8, 5-5f, 5-8, 5-14, 5-26, 9-10f, 10-160f
WPOPB, returning with 4-5, 5-6
WPOPJ 4-5, 4-8, 5-5f, 9-10f, 10-161
WPOPJ, popping with 4-5
WPSH 2-3, 4-1, 4-5, 4-8, 9-8, 9-11, 10-161
WPSH, pushing with 4-5
Wraparound,
  address 1-4f, 5-1, A-2
  narrow stack pointer 5-27
  program counter A-1
Write access 1-9
Write access flag 8-6
Write access violation 5-15
Write command to device map, issuing 7-3
Write data, accessing page to 8-5
Write operation 8-12
Write operation, memory 8-6
Write protection violation 5-17
Write violation fault code D-1
Write, valid 8-8
Writing narrow stack fault handler 5-28
Writing to memory 8-8
Writing wide stack fault handler 5-26
WRSTR 2-9, 4-5, 4-8, 5-8, 7-7, 9-10f, 10-162
WRTN 2-9, 4-4f, 4-8, 5-5ff, 5-13, 9-3f, 9-10f, 10-163
WSALA 2-8, 10-163
WSALM 2-8, 10-163
WSANA 2-8, 10-164
WSANM 2-8, 10-164
WSAVR 2-9, 4-5, 4-8, 5-5ff, 10-164f, D-2
WSAVR, executing 5-13
WSAVS 2-9f, 4-5, 4-8, 5-5ff, 10-165, D-2
WSAVS, executing 5-13
WSB 4-2
WSB,
  load accumulator with 4-4, 10-68
  store accumulator in 4-4, 10-112
WSBI 2-4, 10-166
WSEQ 2-8, 2-23, 10-166
WSEQI 2-8, 2-23, 10-166
WSGE 2-8, 10-167
WSGT 2-8, 10-167
WSGTI 2-8, 10-167
WSKBO 2-8, 10-168
WSKBZ 2-8, 10-168
WSL 4-3
WSL for stack fault, words required beyond 4-8
WSL,
  load accumulator with 4-4, 10-68
  number of words required beyond 4-8

setting bit zero of the 5-25
setting WSP equal to 5-25
store accumulator in 4-4, 10-112
WSLE 2-8, 10-168
WSLEI 2-8, 10-169
WSLT 2-8, 10-169
WSNB 2-8, 2-14, 9-8, 10-169
WSNE 2-8, 10-170
WSNEI 2-8, 10-170
WSP 4-3
WSP equal to WSL, setting 5-25
WSP reserved memory location, updating 5-25
WSP,
decrement double word addressed by 2-9, 4-5, 10-34
increment double word addressed by 2-9, 4-5, 10-65
load accumulator with 4-4, 10-68
load accumulator with double word addressed by 4-5
modifying 4-4
setting bit zero of the 5-25
store accumulator in 4-4, 10-112
store accumulator in double word addressed by 2-3, 4-5
WSSVR 2-9, 4-3, 4-5, 4-8, 5-5f, 5-26, 9-3f, 9-10f, 10-170f, D-2
WSSVR/WRTN, replacing SAVE/RTN with 9-4
WSSVS 2-9f, 4-3, 4-5, 4-8, 5-5ff, 5-26, 9-3f, 9-10f, 10-171, D-2
WSSVS/WRTN, replacing SAVE/RTN with 9-4
WSTB 2-20, 9-8, 10-172
WSTI 2-20, 2-24, 3-3, 5-22, 9-9, 10-172, D-2
WSTI, destination indicator for 5-23
WSTIX 2-20, 3-3, 5-21f, 9-9, 10-173, D-2
WSUB 2-4, 10-173
WSZB 2-8, 2-14, 9-8, 10-174
WSZBO 2-8, 2-14, 9-8, 10-174
WUGTI 2-8, 10-174
WULEI 2-8, 10-175
WUSGE 2-8, 10-175
WUSGT 2-8, 10-175
WXCH 2-3, 10-176
WXOP 4-5, 4-8, 5-5f, 9-10f, 10-176
WXOR 2-13, 10-177
WXORI 2-13, 10-177

X

X instruction prefix 1-11
XCALL 2-9, 4-4ff, 4-8, 5-4ff, 10-177f
XCALL count 4-6
XCH 2-3, 10-178
XCT 5-1f, 10-179
XFAMD 3-6, 9-9, 10-179
XFAMS 3-6, 9-9, 10-180
XFDMD 3-8, 3-14, 9-9, 10-180
XFDMS 3-8, 3-14, 9-9, 10-180
XFLDD 3-4, 9-9, 10-181
XFLDS 3-4, 9-9, 10-181
XFMMD 3-7, 9-9, 10-181
XFMMS 3-7, 9-9, 10-182
XFSMD 3-7, 9-9, 10-182
XFSMS 3-7, 9-9, 10-182
XFSTD 3-4, 9-9, 10-183
XFSTS 3-4, 9-9, 10-183
XJMP 5-2, 7-7, 9-10, 10-183
XJSR 5-4ff, 9-3f, 9-10, 10-183
XLDB 2-20, 9-8, 10-184
XLEF 2-12ff, 2-21, 9-10, 10-184
XLEFB 2-21, 10-184
XNADD 2-4, 10-184
XNADI 2-4, 10-185
XNDIV 2-5, 10-185
XNDO 5-3, 10-185f
XNDSZ 2-9, 9-8, 10-186
XNISZ 2-9, 9-8, 10-186
XNLDA 2-3, 9-8, 10-186
XNMUL 2-5, 10-187
XNSBI 2-4, 10-187
XNSTA 2-3, 9-8, 10-187
XNSUB 2-4, 10-187
XOP0 9-10f, 10-103, 10-188
XOR 2-12f, 10-188
XORI 2-13, 10-189
XPEF 2-21, 4-5, 4-8, 5-7, 10-189
XPEFB 2-21, 4-5f, 4-8, 10-189
XPSHJ 4-5, 4-8, 5-5f, 9-10f, 10-189
XPSHJ, pushing with 4-5
XSTB 2-20, 9-8, 10-190
XVCT 2-9, 4-8, 7-7, 9-10, 10-190
XVCT displacement 7-11
XWADD 2-4, 10-191
XWADI 2-4, 10-191
XWDIV 2-5, 10-191
XWDO 5-3, 10-192
XWDSZ 2-9, 10-192
XWISZ 2-9, 10-192
XWLDA 2-3, 10-193
XWMUL 2-5, 10-193
XWSBI 2-4, 10-193
XWSTA 2-3, 10-193
XWSUB 2-4, 10-194

Z

Z flag of the FPSR 3-10
Z flag, setting (true zero flag) 3-8, 3-13
Zero bit and set bit to one,
skip on 9-6, 9-8, 10-117
wide skip on 9-8, 10-174
Zero bit and set to one, skip on 9-8

Zero bit,
  skip on 9-8
  wide skip on 9-8, 10-174
Zero extend 16-bits to 32-bits 2-2, 10-194
Zero suppression, move numeric with 10-30
Zero,
  crossing to segment 7-9, 7-11
  decrement and skip if 9-8, 10-34
  divisor equals 3-13
  extended decrement and skip if 10-36
  impure E-2
  increment and skip if 10-64
  narrow decrement and skip if 9-8, 10-79, 10-186
  narrow increment and skip if 9-8, 10-79, 10-186
  page 4-2, 5-10f, 5-13, 5-25, 5-27f, 8-11
  redefining wide stack for segment 8-11
  segment 5-8, 5-13, 7-6f, 7-9, 7-11, 8-2, 8-11
  set bit to 9-6f, 10-10f
  set carry to 10-18
  set T to 10-34
  skip if 4-5
  skip on (Z = 1) 3-9, 9-6ff, 10-55, 10-116
  true 3-6, E-2
  wide decrement and skip if 10-87, 10-192
  wide increment and skip if 10-88, 10-192
  wide set bit to 9-8, 10-124
  wide skip on accumulator bit set to 10-168
Zero-extending vector stack limit and pointer 7-9
Zero-extends 1-11
ZEX 2-2, 10-194

Documentation Comment Form

Manual Title _____

Manual No. 014-000704-03

Your Name _____

Your Title _____

Company _____

Street _____

City _____ **State** _____ **Zip** _____

Please help us improve our future publications by answering the questions below. Use the space provided for your comments. Thank you.

	Yes	No
Is this manual easy to read?	☐	☐
Is it easy to understand?	☐	☐
Are the topics logically organized?	☐	☐
Is the technical information accurate?	☐	☐
Can you easily find what you want?	☐	☐
Does it tell you everything you need to know?	☐	☐
Do the illustrations help you?	☐	☐

If you wish to order manuals, contact your sales representative or dealer.

Comments:

Date

Data General

Data General Corporation, Westboro, Massachusetts 01580

014-000704-03