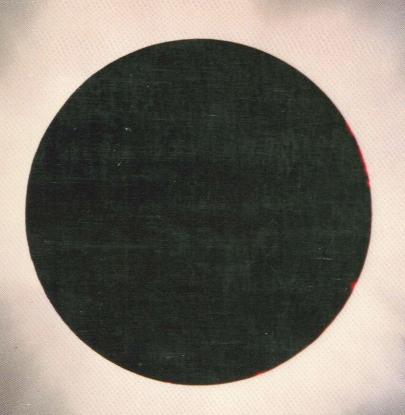
Principles of Operation 32-Bit ECLIPSE® Systems



Programmer's Reference Series

Principles of Operation 32-Bit ECLIPSE® Systems

014-000704-02

Ordering No. 014-000704 *Data General Corporation, 1981, 1982, 1983 All Rights Reserved Printed in the United States of America Revision 02, February 1983

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.

DASHER, DATAPREP, ECLIPSE, ENTERPRISE, INFOS, microNOVA, NOVA, PROXI, SUPERNOVA, PRESENT, ECLIPSE MV/6000, ECLIPSE MV/8000, TRENDVIEW, and MANAP are U.S. registered trademarks of Data General Corporation, and AZ-TEXT, DG/L, ECLIPSE MV/4000, ECLIPSE MV/10000, REV-UP, SWAT, XODIAC, GENAP, DEFINE, CEO, SLATE, microECLIPSE, BusiPEN, BusiGEN and BusiTEXT are U.S. trademarks of Data General Corporation.

Principles of Operation
32-Bit ECLIPSE® Systems
014-000704

Revision History:

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

A vertical bar or an asterisk in the margin of a page indicates substantive change or deletion, respectively from the previous revision.

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.

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-F present the instruction summary, 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 Syst	tem 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 Fixe	ed-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	g-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 I	Management	
	Overview	4-1
	Wide Stack Operations	4-1
	Wide Stack Operations Wide Stack Registers	4-2
	Wide Stack Registers 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 Register Instructions Wide Stack Data Instructions	4-4
	Initializing A Wide Stack	4-6
	Wide Stack Faults	4-7
5 Progra	m Flow Management	
	Overview	5-1
	Program Flow	5-1 5-1
	Related Instruction Groups	5-2
	Execute Accumulator	5-2
	Jump	5-2 5-2
	Skip	5-2 5-2
	Subroutine	5-4
	Transfer Program Control to Another Segment	5-9
	Subroutine Call	5-9

	Subroutine Return	5-13
	Fault Handling	5-13
	Fixed-Point Overflow Fault	5-14
	Floating-Point Overflow and Underflow Faults	5-15
	Decimal and ASCII Data Faults	5-16
	Stack Faults	5-20
<u>6</u>	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 7-10
	Intermediate-Level Interrupt Processing	7-10 7-11
	Final Interrupt Processing	7-11 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
<u>10</u>	Instruction Dictionary	
<u>A</u>	Instruction Summary	
<u>B</u>	Anomalies	
	Wide Instruction Opcodes	B-1
	Program Counter Wraparound	B-1
	Float/Fixed Conversions	B-1
	Address Wraparound	B-1
	C/350 Signed Divide Instructions	B-2
	NIO CPU Instructions	B-2
	Floating-Point Trap	B-2
	Floating-Point Numerical Algorithms	B-2
	C/350 Commercial Faults	B-2
	e, see commercial radius	
<u>C</u> _	ASCII Codes	. 2/5
<u>D</u>	Powers of 2 Table	
<u>E</u>	Fault Codes	
	Protection Faults	E-1
	Stack Faults	E-2
	Decimal/ASCII Faults	E-2
F	Glossary	
<u>r</u>		T 4
	The Addressing Scheme	F-1
	Logical Addresses	F-1
	Segmentation	F-1
	Mapping and Demand Paging	F-1
	Page	F-1
	Page Table	F-1
	Protection	F-2
	The Instruction Set	F-2
	Wide Instructions	F-2
	C/350 Compatibility	F-2
	True and Impure Zero	F-2
	Normalized Format	F-3
	Magnitude	F-3

Illustrations

Figure	Caption	Page
1.1 1.2 1.3 1.4 1.5 1.6 1.7 1.8 1.9 1.10 1.11	Functional components Fixed-point accumulator Floating-point accumulator Wide stack management register format Program counter format Virtual address space Memory reference instruction word addressing formats Memory reference instruction byte addressing formats Byte pointer format Byte addressing Bit pointer format Bit addressing	1-1 1-2 1-3 1-4 1-5 1-7 1-10 1-14 1-14 1-15 1-16
2.1 2.2 2.3 2.4 2.5 2.6 2.7	Fixed-point two's complement data formats ECLIPSE C/350 compatible shift operations Processor status register format Fixed-point logical data formats Explicit data type indicator Packed and unpacked decimal data Decimal arithmetic example	2-2 2-7 2-10 2-12 2-16 2-18 2-24
3.1 3.2	Floating-point data formats Floating-point status register format	3-2 3-10
4.1 4.2 4.3	A typical wide stack Sample code for initializing a wide stack Example of wide stack operations	4-2 4-6 4-7
5.1 5.2 5.3 5.4 5.5 5.6 5.7	Illegal and legal skip instruction sequences DO-loop instruction sequence Subroutine code for an XJSR call Wide stack operations from XJSR and WSSVS instructions Wide stack operations from WRTN instructions Gate array format XCALL or LCALL effective address	5-3 5-4 5-7 5-8 5-8 5-10 5-10
6.1 6.2 6.3 6.4 6.5 6.6	Format of queue descriptor Queue descriptor for an empty queue Data element enqueued into an empty queue Data element enqueued at head of queue Data element enqueued at tail of queue Data element dequeued	6-2 6-3 6-3 6-4 6-4 6-5
7.1 7.2 7.3 7.4 7.5	General I/O instruction format Interrupt sequence Sequence of actions to conclude interrupt service Vector table Device control table (DCT)	7-4 7-8 7-11 7-11 7-12

Figure	Caption	Page
8.1	Segment base register format	8-2
8.2	Page table entry format	8-4
8.3	Indirect and effective logical address formats	8-6
8.4	One-level page table translation	8-8
8.5	Two-level page table translation	8-9
9.1	C/350 program counter format	9-2
9.2	C/350 word addressing format	9-5
9.3	C/350 effective addressing	9-5
9.4	C/350 byte addressing format	9-5
9.5	C/350 byte addressing	9-6
9.6	C/350 bit addressing format	9-6
9.7	BTO, BTZ, SNB, SZB, and SZBO bit addressing	9-7
10.1	ADI example	10-6
	1	10-0
10.2	DAD example	10-0
10.3	DAD example DSB example	_
10.3 10.4	DAD example DSB example DSPA dispatch table structure	10-20
10.3 10.4 10.5	DAD example DSB example	10-20 10-32
10.3 10.4	DAD example DSB example DSPA dispatch table structure Narrow stack, 18-word block Narrow stack, 18-word floating-point return block	10-20 10-32 10-32
10.3 10.4 10.5 10.6 10.7	DAD example DSB example DSPA dispatch table structure Narrow stack, 18-word block Narrow stack, 18-word floating-point return block LDSP dispatch table structure	10-20 10-32 10-32 10-52
10.3 10.4 10.5 10.6 10.7 10.8	DAD example DSB example DSPA dispatch table structure Narrow stack, 18-word block Narrow stack, 18-word floating-point return block LDSP dispatch table structure SBI example	10-20 10-32 10-32 10-52 10-53
10.3 10.4 10.5 10.6 10.7 10.8 10.9	DAD example DSB example DSPA dispatch table structure Narrow stack, 18-word block Narrow stack, 18-word floating-point return block LDSP dispatch table structure SBI example WFPOP 10 double-word	10-20 10-32 10-32 10-52 10-53 10-70
10.3 10.4 10.5 10.6 10.7 10.8 10.9 10.10	DAD example DSB example DSPA dispatch table structure Narrow stack, 18-word block Narrow stack, 18-word floating-point return block LDSP dispatch table structure SBI example WFPOP 10 double-word WFPSH 10 double-word block pushed	10-20 10-32 10-32 10-52 10-53 10-70 10-107 10-132 10-133
10.3 10.4 10.5 10.6 10.7 10.8 10.9 10.10 10.11	DAD example DSB example DSPA dispatch table structure Narrow stack, 18-word block Narrow stack, 18-word floating-point return block LDSP dispatch table structure SBI example WFPOP 10 double-word WFPSH 10 double-word block pushed WXOP return block pushed	10-20 10-32 10-32 10-52 10-53 10-70 10-107 10-132 10-133 10-158
10.3 10.4 10.5 10.6 10.7 10.8 10.9 10.10 10.11 10.12	DAD example DSB example DSPA dispatch table structure Narrow stack, 18-word block Narrow stack, 18-word floating-point return block LDSP dispatch table structure SBI example WFPOP 10 double-word WFPSH 10 double-word block pushed WXOP return block pushed XOP0 return block	10-20 10-32 10-32 10-52 10-53 10-70 10-107 10-132 10-133
10.3 10.4 10.5 10.6 10.7 10.8 10.9 10.10 10.11	DAD example DSB example DSPA dispatch table structure Narrow stack, 18-word block Narrow stack, 18-word floating-point return block LDSP dispatch table structure SBI example WFPOP 10 double-word WFPSH 10 double-word block pushed WXOP return block pushed	10-20 10-32 10-32 10-52 10-53 10-70 10-107 10-132 10-133 10-158

Tables

Table	Caption	Page
1.1 1.2	Effective addressing Faults	1-12 1-10
2.1 2.2 2.3 2.4 2.5 2.6	Range of 16- and 32-bit two's complement numbers Fixed-point precision conversion Fixed-point data movement instructions Fixed-point addition instructions Fixed-point subtraction instructions Fixed-point multiplication instructions	2-2 2-2 2-3 2-4 2-4 2-5 2-5
2.7 2.8 2.9 2.10 2.11 2.12	Fixed-point division instructions Initializing carry instructions Fixed-point skip on condition instructions Fixed-point increment or decrement word and skip instructions PSR manipulation instructions Logical instructions	2-3 2-6 2-8 2-9 2-9 2-1.
2.13 2.14 2.15 2.16 2.17	Logical shift instructions Fixed-point logical skip instructions Explicit data types Sign and number combination for unpacked decimal Nonsign-positioned numbers for unpacked decimal	2-14 2-14 2-17 2-19 2-19
2.18 2.19 2.20 2.21 2.22	Fixed-point byte movement instruction Fixed-point to floating-point conversion and store instructions Load effective word and byte address instructions Edit subprogram instructions Hex shift instructions	2-20 2-20 2-21 2-21 2-22
3.1 3.2 3.3 3.4 3.5 3.6 3.7 3.8 3.9	Floating-point binary conversion instructions Floating-point decimal conversion instructions Floating-point data movement instructions Floating-point addition instructions Floating-point subtraction instructions Floating-point multiplication instructions Floating-point division instructions Floating-point skip on condition instructions Floating-point status instructions	3-3 3-3 3-4 3-6 3-7 3-8 3-9 3-10
4.1 4.2 4.3 4.4 4.5	Wide stack register instructions Wide stack double-word access instructions Wide stack return block instructions Standard wide return block Multiword stack instructions	4-4 4-5 4-5 4-6 4-8
5.1 5.2 5.3 5.4 5.5	Jump instructions Skip instructions Subroutine instructions Sequence of subroutine instructions Standard wide return block	5-2 5-3 5-5 5-5 5-7

Table	Caption	Page
5.6 5.7 5.8	Segment transfer instructions Faults Fixed-point fault return block	5-9 5-13 5-15
5.8 5.9 5.10 5.11 5.12 5.13 5.14 5.15 5.16 5.17 5.18 5.19 5.20	Wide floating-point fault return block Narrow floating point fault return block Decimal and ASCII fault codes Wide return block for decimal data (type 1) fault Wide return block for ASCII data (type 2) fault Wide return block for ASCII data (type 3) fault Narrow return block for decimal data (type 1) fault Narrow return block for ASCII data (type 2) fault Narrow return block for ASCII data (type 2) fault Narrow return block for ASCII data (type 3) fault Wide stack fault return block Wide stack fault return block Narrow stack fault return block	5-16 5-16 5-17 5-18 5-18 5-19 5-19 5-19 5-21 5-21 5-21
6.1 6.2 6.3	Data element with user data following links Data element with user data preceding links Queue instructions	6-2 6-2 6-5
7.1 7.2 7.3 7.4	I/O instructions for data/BMC maps General I/O instructions Device flags for general devices Device flags for skip instruction	7-3 7-3 7-4 7-5
8.1 8.2 8.3 8.4 8.5	Instructions that manipulate referenced and modified flags System identification instructions Priority of protection violation faults Fault return block Protection fault codes	8-11 8-11 8-12 8-13 8-14
9.1 9.2 9.3 9.4 9.5	Alternations to C/350 subroutines C/350 Fixed-point computing instructions C/350 Floating-point computing instructions C/350 program flow management instructions C/350 stack management instructions	9-4 9-8 9-9 9-10 9-11
E.1 E.2 E.3	Protection Violation Fault Codes Wide Stack Fault Codes Decimal and ASCII Fault Codes	E-1 E-2 E-2

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.

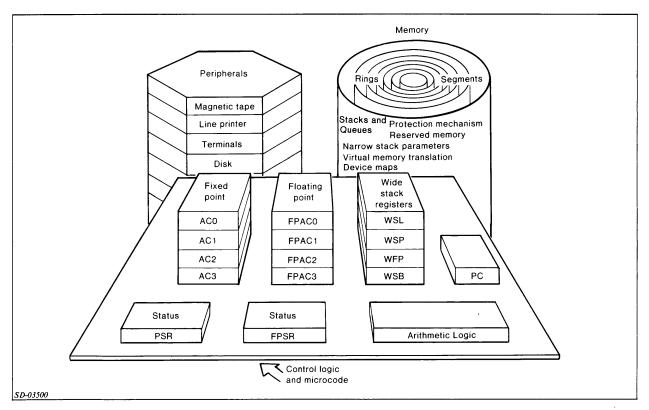


Figure 1.1 Functional components

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).

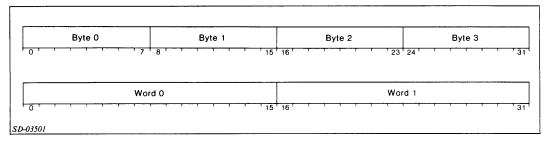


Figure 1.2 Fixed-point accumulator

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, a fixed-point overflow fault flag, and an interrupt resume 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 interrupt resume flag reports an instruction status to the processor.

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).

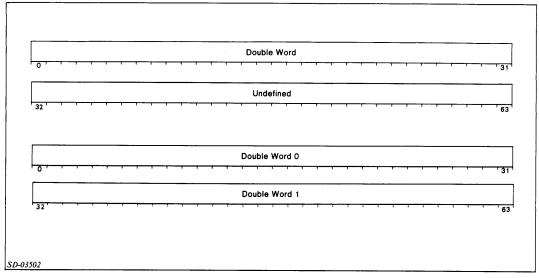


Figure 1.3 Floating-point accumulator

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.

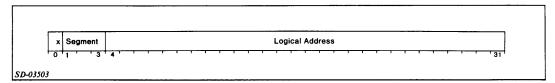


Figure 1.4 Wide stack management register format

where

x Bit 0 is reserved for future use.

Segment Bits 1-3 specify the segment location of the stack.

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

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.

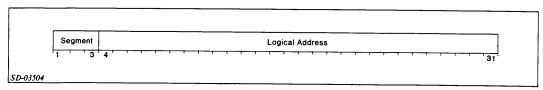


Figure 1.5 Program counter format

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.

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.

			Displacement Range	
Address Mode	Index	Intermediate Logical Address*	Prefix**	Octal Words (decimal)
Absolute	00	D		0 to 377 (0 to 255)
		D	×	0 to 177777 (0 to 65,535)
		D	L	0 to 3777777777 (0 to 2,147,483,647)
PC Relative	01	PC ± D		- 200 to + 177 (- 128 to + 127)
		PC+n±D	×	- 100000 to + 77777 (- 32,768 to + 32,767)
		PC+n±D	L	- 20000000000 to + 17777777777 (- 1,073,741,824 to + 1,073,741,823)
AC2 Relative	10	AC2 ± D		- 200 to + 177 (- 128 to + 127)
		AC2±D	×	- 100000 to + 77777 (- 32,768 to + 32,767)
		AC2±D	L - 20000000000 to + 1777777777 (- 1,073,741,824 to + 1,073,741,823)	
AC3 Relative	11	AC3±D		- 200 to + 177 (- 128 to + 127)
		AC3±D	×	- 100000 to + 77777 (- 32,768 to + 32,767)
		AC3±D	L	- 20000000000 to + 1777777777 (- 1,073,741,824 to + 1,073,741,823)

Table 1.1 Effective addressing

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.

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.)

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

^{**}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.

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.

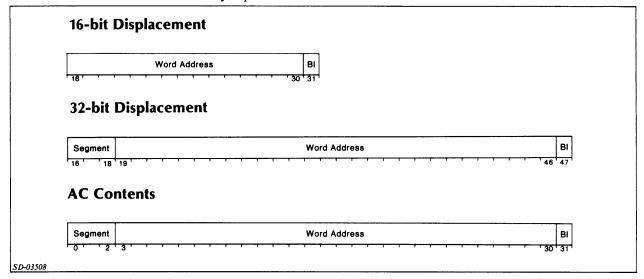


Figure 1.9 Byte pointer format

where

Segment Word Address The segment field identifies the current or an outward memory segment.

The word address field identifies a 16-bit word in the memory segment.

ΒI

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).

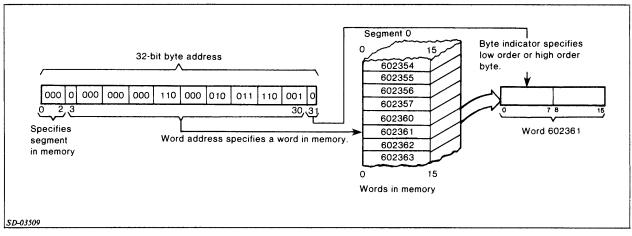


Figure 1.10 Byte addressing

Bit

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

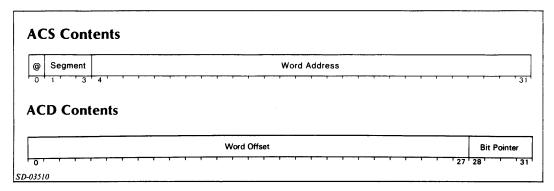


Figure 1.11 Bit pointer format

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 Pointer	The bit pointer 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	Туре
Protection violation Nonresident page Stack operation Fixed-point computation Floating-point computation	Privileged Privileged Nonprivileged Nonprivileged 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. Refer to the Program Flow Management chapter for further details on servicing a nonprivileged fault.

A privileged fault occurs when the processor detects a protection violation or page fault, such as an unauthorized I/O operation or a nonresident memory page reference. 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 Memory and System Management chapter for further details on accessing the protection mechanisms and servicing a privileged fault.

Summary

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

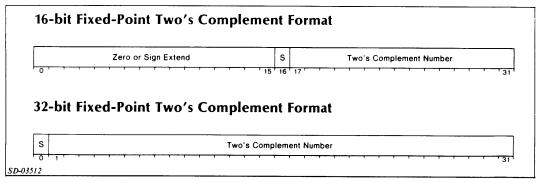


Figure 2.1 Fixed-point two's complement data formats

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 zero 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	O to 65,535	0 to 4,294,967,295
Signed	-32,768 to	-2,147,483,648 to
	+32,767	+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 or zero 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

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 and of the destination accumulator.

^{*}ECLIPSE C/350 compatible instruction

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

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.

^{*}ECLIPSE C/350 compatible 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

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.

^{*}ECLIPSE C/350 compatible instruction

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.

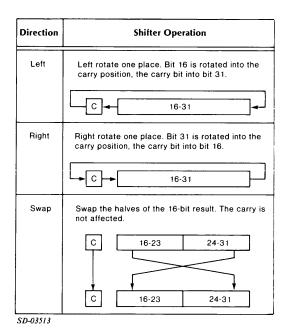


Figure 2.2 ECLIPSE C/350 compatible shift operations

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

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.

^{*}ECLIPSE C/350 compatible instruction

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

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
ВКРТ	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 ACO
PBX	Pop block and execute
SPSR	Store PSR from ACO
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

^{*}ECLIPSE C/350 compatible instruction

Figure 2.3 shows the format of the processor status register.

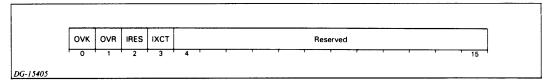


Figure 2.3 Processor status register format

NOTES: The IRES and IXCT bits are for hardware use. Do not modify the state of either bit; 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.

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 4 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.

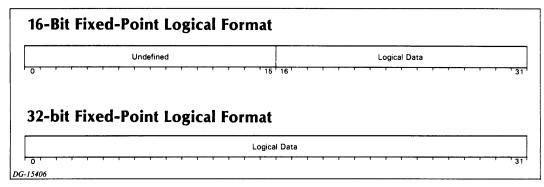


Figure 2.4 Fixed-point logical data formats

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

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

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.

^{*}ECLIPSE C/350 compatible instruction

^{*}ECLIPSE C/350 compatible instruction

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

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 the move and convert instructions (such as WEDIT and WLDI), the processor requires (in AC1) an explicit data type indicator, as shown in Figure 2.5.

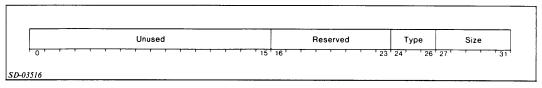


Figure 2.5 Explicit data type indicator

where

Size

Reserved The reserved field indicates that DGC reserves bits 16-23 for future use.

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

The size field indicates, as an unsigned integer, the length of the data. The following list explains the data type and corresponding size specification.

•	
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 2 bytes
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

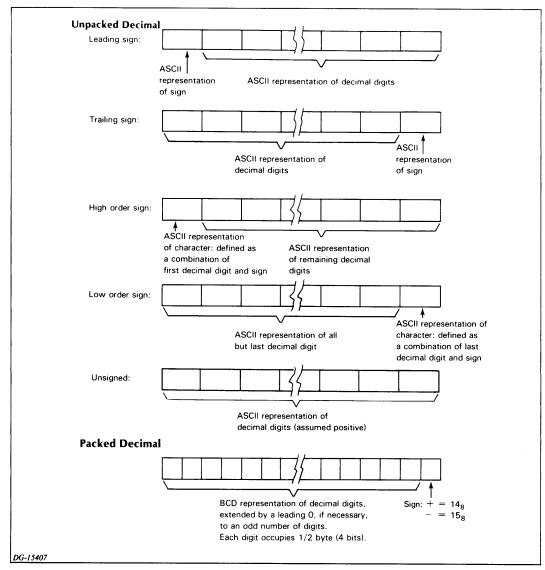


Figure 2.6 Packed and unpacked decimal data

Digit and Sign	Char	CII acter code)	Digit and Sign	Char	SCII acter code)	Digit and Sign	Char	CII acter code)
0+	space	(040)	5+	5	(065)	1-	J	(112)
0+	+	(053)	5+	E	(105)			
0+	{	(173)				2-	ĸ	(113)
0+	0	(060)	6+	6	(066)			
			6+	F	(106)	3-	L	(114)
1+	1	(061)						
1+	Α	(101)	7+	7	(067)	4-	М	(115)
			7+	G	(107)			
2+	2	(062)				5-	N	(116)
2+	В	(102)	8+	8	(070)			
			8+	Н	(110)	6-	0	(117)
3+	3	(063)						
3+	С	(103)	9+	9	(071)	7-	Р	(120)
			9+	ļı .	(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		Character al 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	
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 WLDIX WSTI	Convert a decimal and load into FPAC Convert a decimal, extend, and load it into four FPACs 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 LPEF	Load effective byte address Push address
LPEFB WMOVR	Push byte address Wide move right (convert byte pointer to word pointer)
XLEF XLEFB	Load effective address 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 (DAD) and subtract (DSB) 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.

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 2.22 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 2.22 Hex shift instructions

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.

^{*}ECLIPSE C/350 compatible 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 bit 0 of AC1 with the results. You can then use the wide skip on accumulator bit instructions (WSKBO and WSKBZ) to test the condition.

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. 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.

```
TITL
                                                   DECIMAL
                                            .ENT
                                                    START
                                            NREL
                                            : CONSTANTS
                                            ENABLE SWORD
                                    CON:
                                            .TXT
                                                    **CONSOLE*
                                                                     :GENERIC CONSOLE NAME
                                                                    :TYPE 4 AND 3 DECIMAL DIGITS
:RESERVE 5 WORDS FOR NUMBER BUFFER
                                    FCON:
                                            202
                                    IBUF :
                                            .BLK 5
                                            PARAMETER PACKETS
                                    :READ CONSOLE PACKET TO OPEN. READ. & WRITE
                                    CONSOLE: .BLK 22
.LOC CONSOLE+?ISTI
                                                                         :DATA SENSITIVE I/O
                                              ?RTDS+0F10
                                              .LOC CONSOLE+?ISTO
                                              .LOC CONSOLE+?IMRS
                                              .LOC CONSOLE+?IBAD
                                                                         .?IBAD CONTAINS BYTE POINTER TO DATA PACKET
                                              .LOC CONSOLE+?IRCL
                                              -1
                                              180.
                                              .LOC CONSOLE+?IRNW
                                              0
                                              .ENABLE DWORD
                                              .LOC CONSOLE+?IDEL
                                              .LOC CONSOLE+PETSP
                                              .LOC CONSOLE+PETFT
                                              0
                                              .LOC CONSOLE+PETLT
                                     :END OF CONSOLE PACKET
                                                                          :OPEN CONSOLE TO READ AND WRITE
                                    START: ?OPEN CONSOLE
                                              ?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
                                                                           :SINGLE PRECISION FLOATING-POINT ADD
                                              FAS
                                                     0.0
                                              XNLDA 1.FCON
                                              XNLDA 3.CONSOLE+?IBAD
                                    AGAIN:
                                              WSTI
                                              INC
                                                      1, 1, SZC
                                                                           :INC BYTE COUNT AND SKIP IF WSTI TRUNCATES
                                                      AGAIN
                                                                           :REPEAT WSTI
                                                                          :DISPLAY THE SUM ON THE CONSOLE
                                              ?WRITE CONSOLE
                                              ?CLOSE CONSOLE
                                                                          :CLOSE THE CONSOLE
                                                                          :RETURN TO CLI
                                              ?RETURN
                                               .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.

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) and the floating-point status register (FPSR).

Data Formats

The floating-point arithmetic 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.

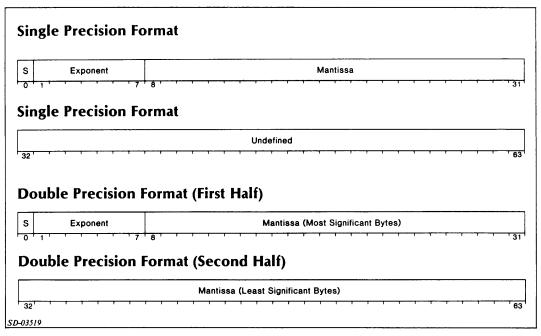


Figure 3.1 Floating-point data formats

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

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 (ACO 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 ACO 16-31)
FSCAL *	Scale floating point
WFFAD	Wide fix from FPAC
WFLAD	Wide float from AC

Table 3.1 Floating-point binary conversion instructions

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 WLDIX WSTI WSTIX	Convert a decimal and load into FPAC Convert a decimal, extend, and load it into four FPAC's Convert FPAC data and load into memory 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.

^{*}ECLIPSE C/350 compatible instruction

Instruction	Operation
FMOV *	Move floating point (FPAC to FPAC)
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

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.

*

^{*}ECLIPSE C/350 compatible instruction

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 7F₁₆ inclusive, the intermediate result becomes the final result (without change).
- Equal to 80₁₆, 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 81₁₆ to FF₁₆ inclusive, the processor adds 1₁₆ 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 1₁₆ 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 * FAS * LFAMD LFAMS XFAMD XFAMS	Add double (FPAC to FPAC) Add single (FPAC to FPAC) Add double (memory to FPAC) Add single (memory to FPAC) Add double (memory to FPAC) 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 \neq facd). Table 3.5 lists the floating-point subtract instructions.

Instruction	Operation
FSD *	Subtract double (FPAC from FPAC)
FSS *	Subtract single (FPAC 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

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)
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

^{*}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 DVZ error flag to one, 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 0₁₆ 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 64₁₀ 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)
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

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.

^{*}ECLIPSE C/350 compatible instruction

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

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.

^{*}ECLIPSE C/350 compatible instruction

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 3.9 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 3.9 Floating-point status instructions

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

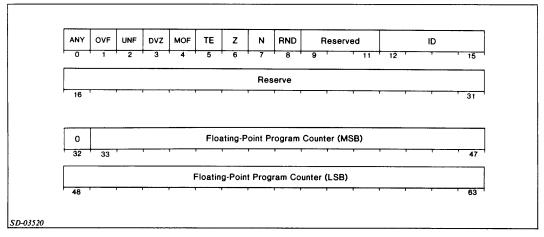


Figure 3.2 Floating-point status register format

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, DVZ, 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.

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.

^{*}ECLIPSE C/350 compatible instruction

DVZThe DVZ bit is the mantissa divide by zero flag.

> The processor sets the DVZ flag while attempting to execute a floatingpoint divide instruction with a divisor equal to zero. The processor then aborts the division and the operands remain unchanged.

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.

> Unlike the OVK mask, the processor does not save or restore the status of the TE mask when going to or returning from a subroutine or fault handler.

> 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.

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 16-32 are processor specific.

Floating-The floating-point program counter contains the address of the Point floating-point instruction and identifies the instruction, which Program causes a floating-point error. Counter

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.

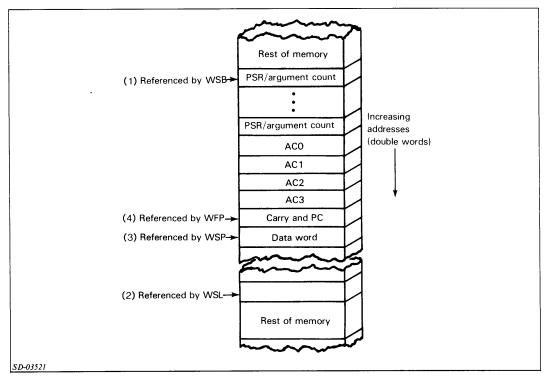


Figure 4.1 A typical wide stack

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 48 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 to two less than 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
ВКРТ	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	ACO
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 256₁₀ through 355₁₀. The processor detects a stack overflow 17 double words before the actual end of the stack.

	NREL			
BASE: .E				words for the wide stack
ENDZ: .E	BLK :	34.	: Reserve 34	words for wide stack end zone
	•			
XI	_EF 0	, BASE		
			: Initialize	WSB
XI	_EF 0	, ENDZ	: Initialize	WSL for a stack
S.	TASL 0		: overflow	when WSP = BASE+66
XI	_EF 0	,BASE-2		
S.	TASP 0		: Initialize	WSP
S.	TAFP 0		; Initialize	WFP
X	PEFB B	YTZ*2		and store the byte address
			for BYTZ	

Figure 4.2 Sample code for initializing a wide stack

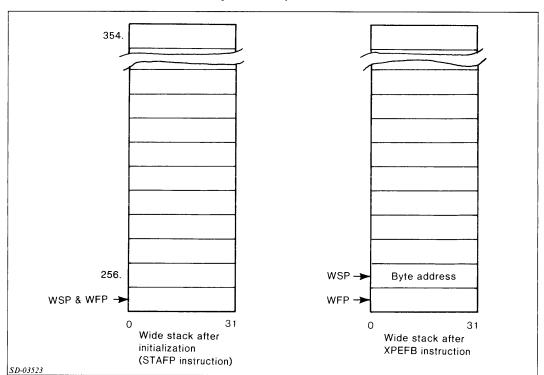


Figure 4.3 illustrates the result of executing the assembler code in Figure 4.2. The XPEFB instruction calculates and pushes a byte address onto the stack.

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 377777777778 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 2000000000_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.

		Double Words		
Instruction	Description	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	
ВКРТ	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	WSSVS Wide special save/set OVK		11	
WXOP	WXOP Extended operation		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 LJMP WBR XJMP	Dispatch Jump (with long displacement) Branch (PC relative jump) 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.

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

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 (for skipping the WBR instruction) 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.

^{*}ECLIPSE C/350 compatible instruction

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

```
WSUB
                                2.2
                                                      · Zero AC2
                       XWSTA
                                2.TMP
                                                      : Initialize loop variable to 0
           LOOP:
                       WI DAT
                                11,2
                                                      ; Initialize (AC2) to 11
                       XWDO
                                2, TERM, TMP
                                                      ; Increment TMP and compare to AC2
                                                      ; Begin here to perform the DO-loop
                       WBR
                                LOOP
                                                      ; Continue the DO-loop
           TERM:
                                                      ; End of the DO-loop when TMP > AC2
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
ВКРТ	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
ВКРТ	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

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.

^{*}Use the BKPT/WPOPB instruction sequence when removing the BKPT instruction before returning 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 ACO. 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. 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	ACO
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.

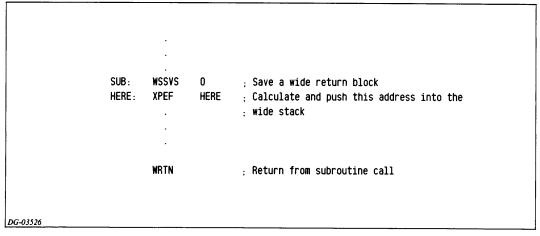


Figure 5.3 Subroutine code for an XJSR call

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.

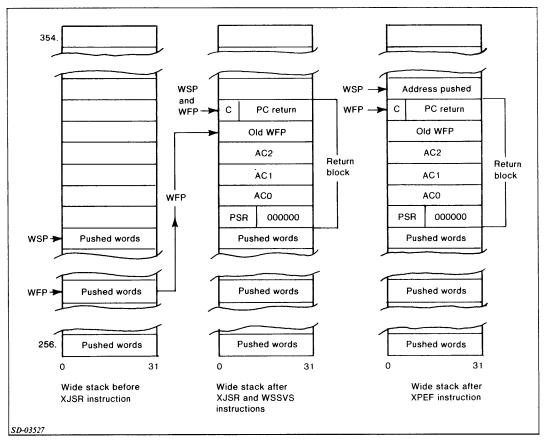


Figure 5.4 Wide stack operations from XJSR and WSSVS instructions

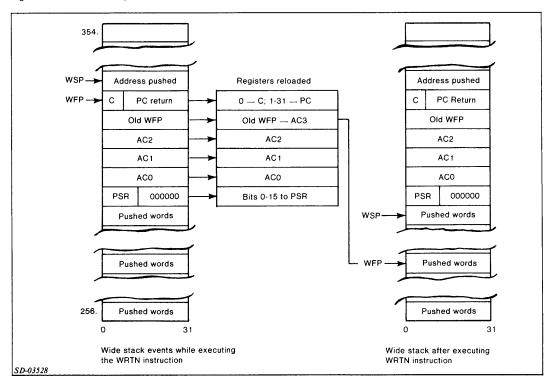


Figure 5.5 Wide stack operations from WRTN instruction

Transfer 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.

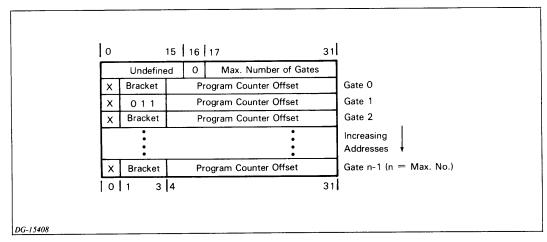


Figure 5.6 Gate array format

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 0112, 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.

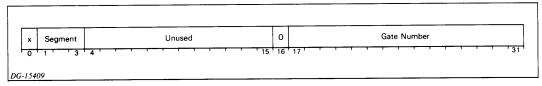


Figure 5.7 XCALL or LCALL effective address

where

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.

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.

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.

- 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

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.

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.

- 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.
- 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.
- 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	Туре
Protection violation	Privileged
Nonresident page	Privileged
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 E 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.

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 5.8 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 of the instruction address causing the fault

Table 5.8 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 Overflow and Underflow Faults

The processor detects a floating-point overflow or underflow, when attempting division by zero, 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 fault flag (OVF, UNF, DVZ, and MOF) and the ANY flag to one.

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 5.9 shows the wide return block and Table 5.10 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 of the instruction address causing the fault	

Table 5.9 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 of the instruction address causing the fault	

Table 5.10 Narrow floating-point fault return block

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.

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

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 5.11 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 Retur	ned in AC1			
Narrow	Wide	Return Block Type	Faulting Instruction	Meaning
000000	100000	2	EDIT, WEDIT	An invalid digit or alpha- betic character encoun- tered during execution of one of the following subopcodes: DMVA, DMVF, DMVN DMVO, DMVS
000001	100001	1	LDIX, STIX	Invalid data type (6 or
		3	EDIT, WEDIT WLDIX, WSTIX	7)
000002	100002	2	EDIT, WEDIT	DMVA or DMVC subopcode with source data type 5; AC2 con- tains 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 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	1	WLSN, WLDI, LSN, LDI LDIX, WLDIX	Sign code is invalid for this data type
		3	EDIT, WEDIT	
000007	100007	1	WLSN, WLDI, WLDIX, LSN LDI, LDIX	Invalid digit

Table 5.11 Decimal and ASCII fault codes

Wide Fault Return Blocks

Tables 5.12 through 5.13 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	ACO 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 5.12 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	ACO 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 5.13 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
5-6	AC1 bits 0-31 original descriptor
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

Table 5.14 Wide return block for ASCII data (type 3) fault

Narrow Fault Return Blocks

Tables 5.15 through 5.17 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.

Contents	
ACO bits 16-31 unchanged	
AC1 bits 16-31 contain original descriptor	
AC2 bits 16-31 contain original source indicator (destination indicator for WSTI or STIX)	
AC3 bits 16-31 undefined	
PC bits 17-31 of the decimal instruction address causing the fault	

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

Word Number in Block Pushed	Contents	
1-4	reserved (for ECLIPSE compatibility)	
5	ACO 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 5.16 Narrow return block for ASCII data (type 2) fault

Word Number in Block Pushed	Contents	
1	ACO 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 5.17 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 5.18).

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 5.19).
- 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 5.18 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 5.19 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 401₈.

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 5.20).

The return address word in the narrow return block points to the next instruction the processor executes after servicing the fault.

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 5.20 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.
- Take any other appropriate action, such as allocating more stack space or 3. 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 head. Conversely, data elements can be dequeued anywhere in the queue, not just at the tail.

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 Second double word	Forward link Backward link
Next n double words	User information

Table 6.1 Data element with user data following links

Position in Data Element	Contents
First <i>n</i> 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).

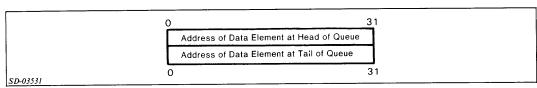


Figure 6.1 Format of queue descriptor

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.

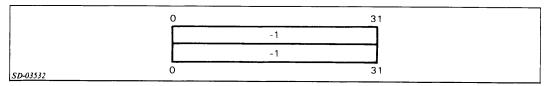


Figure 6.2 Queue descriptor for an empty queue

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.

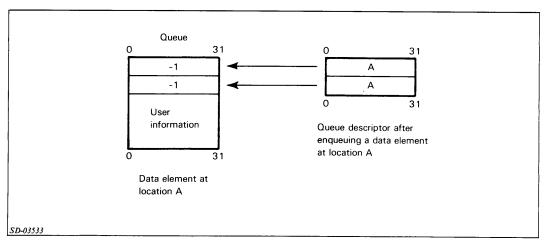


Figure 6.3 Data element enqueued into an empty queue

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.

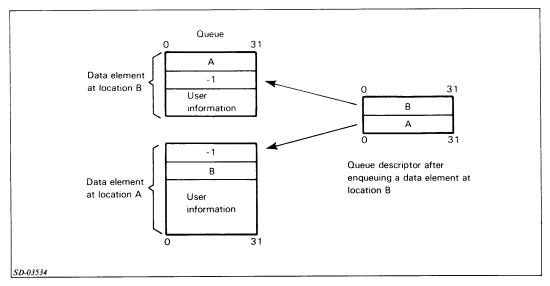


Figure 6.4 Data element enqueued at head of queue

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.

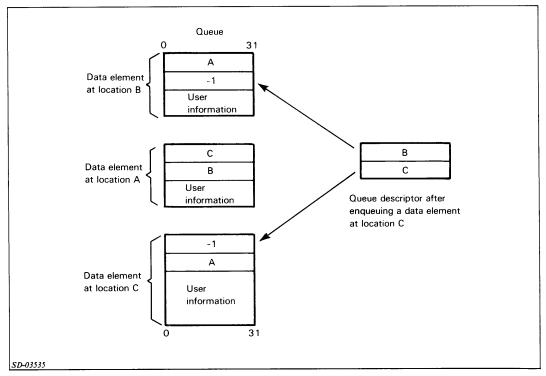


Figure 6.5 Data element enqueued at tail of queue

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.

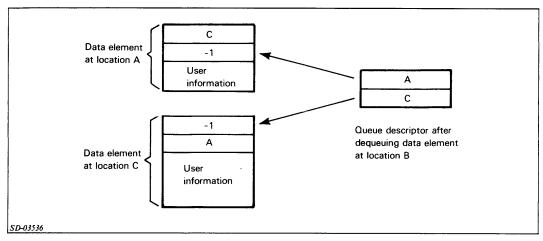


Figure 6.6 Data element dequeued

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 4 instructions perform queue 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		

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)		
DO3[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		
SKPt *	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.

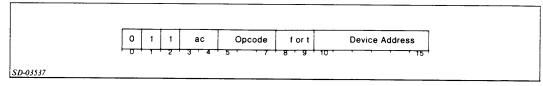


Figure 7.1 General I/O instruction format

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	I/O		CPU
	for f 8 9	Busy	Done	ION
(option omit-				
ted)	0 0	No effect	No effect	No effect
s	0 1	Set to a 1	Set to a 0	Set to a 1
С	1 0	Set to a 0	Set to a 0	Set to a 0
Р	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	1/0	СРИ
BN BZ DN DZ	0 0 0 1 1 0 1 1	Test for BUSY = 1 Test for BUSY = 0 Test for DONE = 1 Test for DONE = 0	Test for ION = 1 Test for ION = 0 Test for power fail = 1 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.

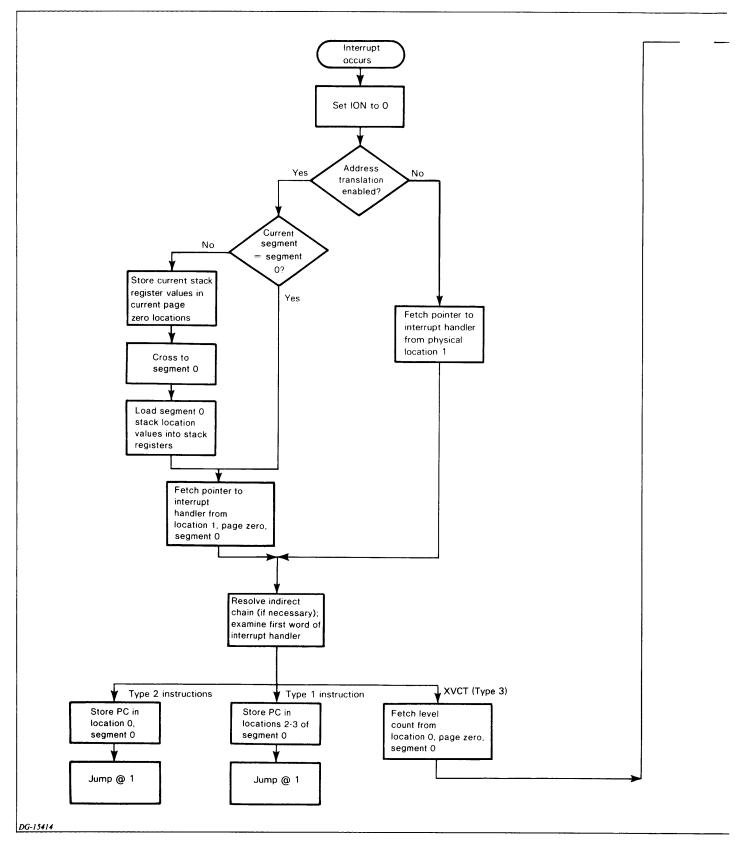
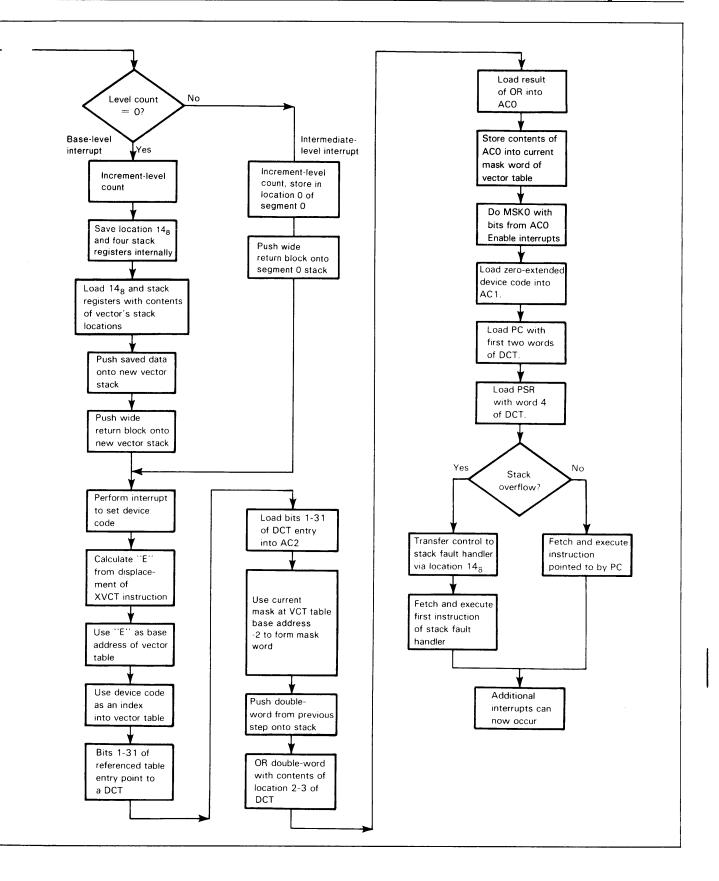


Figure 7.2 Interrupt sequence (continues)



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.

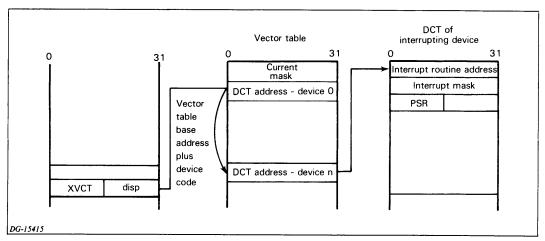


Figure 7.3 Sequence of actions to conclude interrupt service

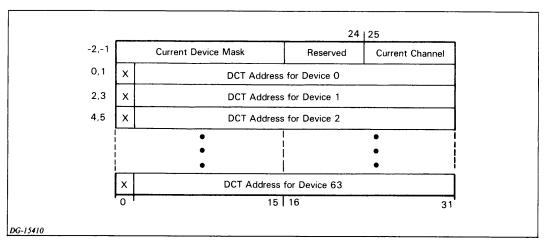


Figure 7.4 Vector table

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.

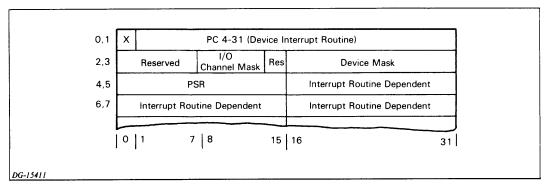


Figure 7.5 Device control table (DCT)

- 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 Fault section 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.

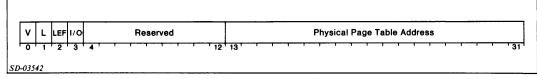


Figure 8.1 Segment base register format

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 zero for a LEF instruction interpretation.

The processor executes the instruction as a LEF instruction.

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

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

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.

DGC reserves bits (4-12) for internal DGC use. Refer to the specific

functional characteristics manual for additional information.

Physical Page Table Address

The physical page table address (bits 13-31) specifies the 19 most significant bits of the physical address for the first page table entry. (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.

I/O

Reserved

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 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.

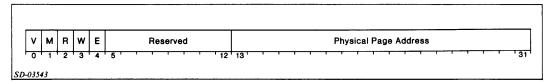


Figure 8.2 Page table entry format

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.

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.

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.

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.

W

E

R

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 The physical page address (bits 13-31) identifies a page in memory.

Address 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.

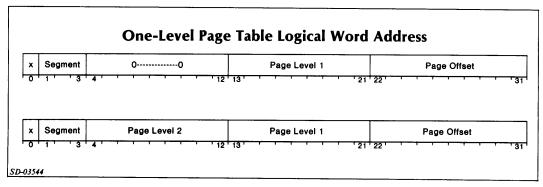


Figure 8.3 Indirect and effective logical address formats

where

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 Fault section 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 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.

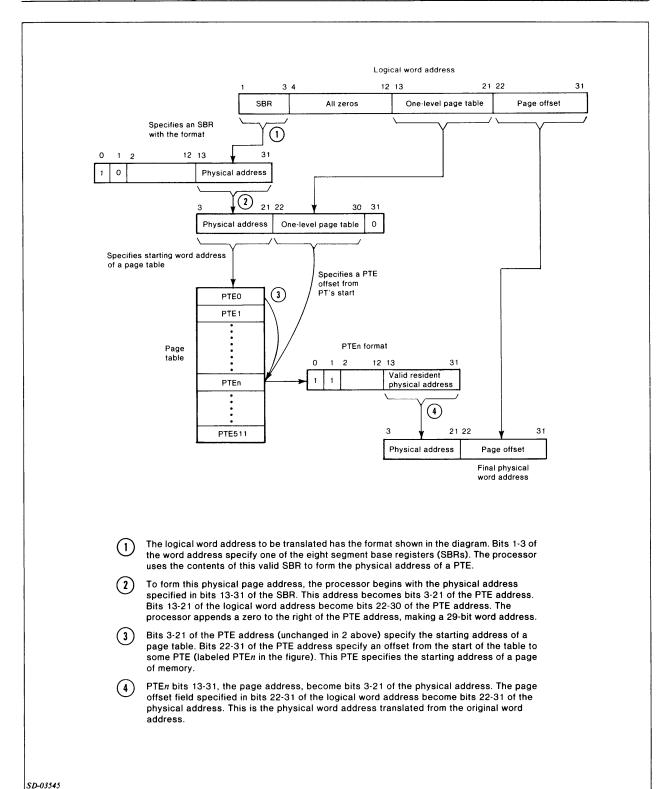


Figure 8.4 One-level page table translation

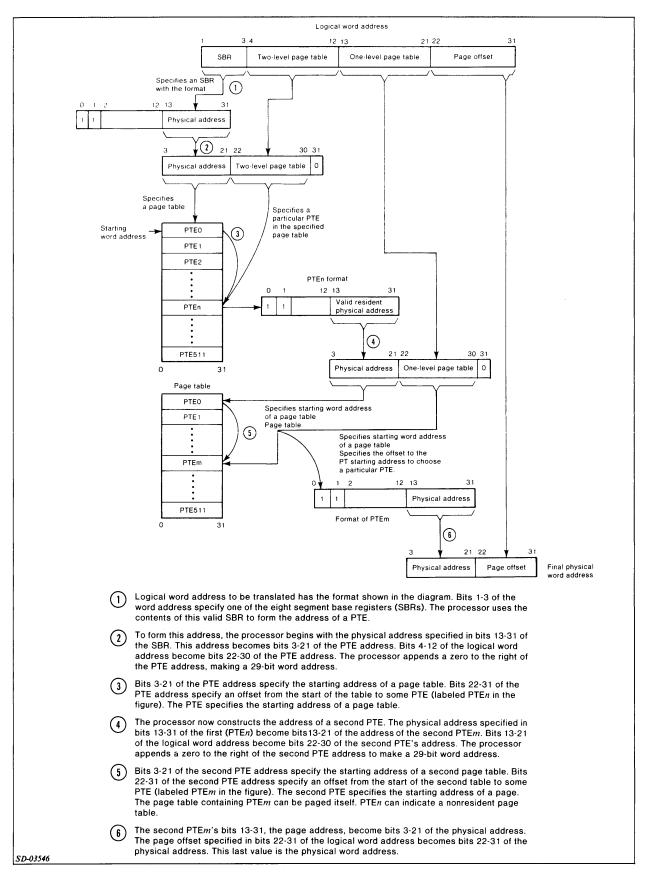


Figure 8.5 Two-level page table translation

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 or refers to a location that requires a two-level page table when only a one-level page table is allocated, 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.

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.

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 (ACO, bits 0-31)	
LCPID	Load CPU identification (ACO, bits 0-31)	
NCLID	Narrow load CPU identification (ACO-AC2, bits 16-31)	

Table 8.2 System identification instructions

Protection Violations

As the Program Flow Management chapter explains, the processor performs certain checks on the operation and on the data while executing an instruction. If the processor detects an error, a privileged or nonprivileged fault occurs before executing the next instruction. Refer to the Program Flow Management chapter for information on nonprivileged faults.

Since an operation could produce multiple protection violations, the processor imposes priorities on the faults. The processor services the highest priority fault and ignores lower priority faults, when two or more occur. Table 8.3 lists the protection violation faults in the order of priority. For instance, the processor services a level 2 priority and ignores a level 4 priority, when both occur simultaneously.

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

Table 8.3 Priority of protection violation faults

When the processor detects a fault, it performs a segment crossing to segment 0 (if the fault occurs in segment 1 to 7) and jumps to the protection violation fault handler through the indirect pointer in reserved memory. The initial and indirect pointers to the protection violation fault handler are 16 bits. Levels of indirection, if any, occur within segment 0.

If a protection violation fault occurs while handling a nonprivileged fault, the processor aborts the nonprivileged fault and processes the protection violation fault. The return block pushed onto the stack for the protection violation fault is undefined, as are the contents of AC0 and AC1.

If an I/O interrupt request occurs, the processor executes the first instruction of the protection violation fault handler before servicing the interrupt request.

To service a protection violation fault, the processor

- 1. Crosses to segment 0 if the current segment is 1 to 7.
 - a. 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.
 - b. 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.

2. Pushes a fault return block onto the stack in segment 0.

Table 8.4 shows the fault return block.

The PSR is then set to zero.

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 8.4 Fault return block

3. Initializes AC0 and AC1.

- a. Sets AC0 equal to the address of the instruction (offending PC) causing the fault.
- b. Sets AC1 equal to a value identifying the fault.

Table 8.5 lists the protection fault codes.

- c. Sets AC2 equal to the specific address (offending address) that caused the reference problem, if applicable (bit 0 is undefined). See Table 8.5.
- 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 original fault.

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

5. Jumps to the fault handler and executes the first instruction.

To transfer program control, the processor refers to a pointer in reserved memory, that contains the starting address of the fault handler. Each segment maintains a set of reserved memory locations. Refer to the specific functional characteristics manual for further information on reserved memory.

*

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

Table 8.5 Protection fault codes

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 ignores bits 0-15. 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).

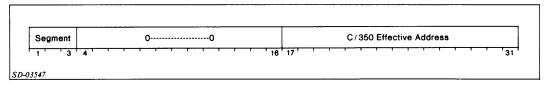


Figure 9.1 C/350 program counter format

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.

- 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 (177777₈), 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-referenced 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	pushed stack block, requiring the Assembler to recalculate the negative	
Change a subroutine (to save the 31-bit PC) that a routine refers to with a JSR through page zero	1 ' -	

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. (Appendix A identifies the C/350 instructions supported on the 32-bit processor.) 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).

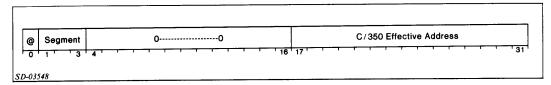


Figure 9.2 C/350 word addressing format

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.

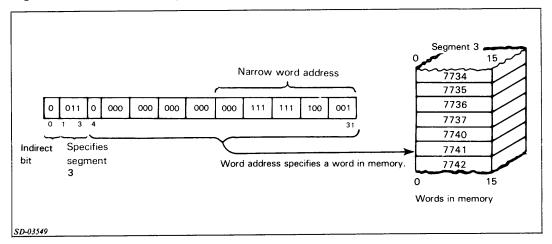


Figure 9.3 C/350 effective addressing

To refer to a byte, with a C/350 memory reference instruction, the processor forms a byte address (see Figure 9.4).

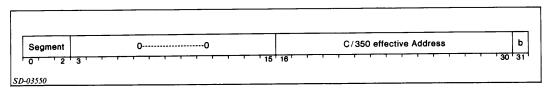


Figure 9.4 C/350 byte addressing format

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).

Segment 0 Byte indicator specifies 15 low order or high order Narrow byte address 4030 4031 4032 000 000 000 000 000 000 100 റററ 011 110 4033 15 16 30 31 4034 4035 Specifies Word 4036 4036 seament () Word address specifies a word in memory 4037 in memory 4040 Words in memory SD-03551

Figure 9.5 illustrates C/350 byte addressing.

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 word pointer and bit pointer from the contents of two accumulators. The word pointer consists of an effective address (in the ACS accumulator) and a word offset (in the ACD accumulator). The bit pointer 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.

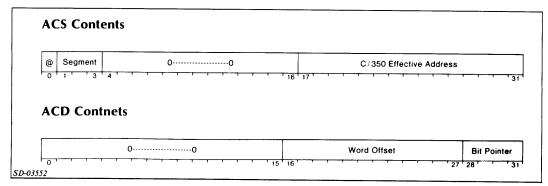


Figure 9.6 C/350 bit addressing format

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 pointer, which specifies the bit position (0-15) in the final word.

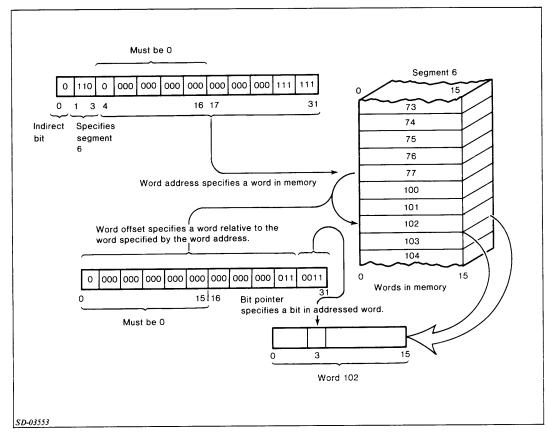


Figure 9.7 BTO, BTZ, SNB, SZB, and SZBO bit addressing

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

C/350 Instruction	Operation	Equivalent Instruction
BAM	Block add and move	_
BLM	Block move	WBLM
вто	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
	1	l .

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

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.

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.

C/350 Instruction	Operation	Equivalent Instruction
DSPA	Dispatch	LDSP
ЕЈМР	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.

C/350 Instruction	Operation	Equivalent Instruction
MSP	Modify stack pointer	WMSP
POP	Pop multiple accumulators	WPOP
POPB	Pop block and execute (return from XOPO)	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 10 Instruction Dictionary

Uppercase or boldface characters indicate a literal argument in

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.

UPPERCASE

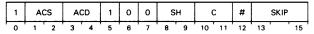
• Functional description of each instruction.

The Instruction Dictionary uses the following conventions and abbreviations.

and/or boldface	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

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]

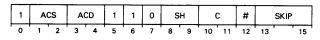
The processor can skip the next instruction if the condition test is true. The following lists gives the test conditions, bits 13 to 15, and the operation.

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]	Bi 10-	ts -11	Operation
omitted	0	0	Leave CARRY unchanged
Z	0	1	Initialize CARRY to 0
0	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]

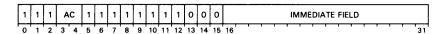
The processor can skip the next instruction if the condition text is true. The following lists gives the test conditions, bits 13 to 15, and the operation.

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



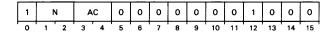
Adds a signed integer in the range of -32,768 to +32,767 to the contents of an accumulator.

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 177775₈. After the instruction ADI 4,2 is executed, AC2 contains 000001₈ and carry is unchanged. (See Figure 10.1.)

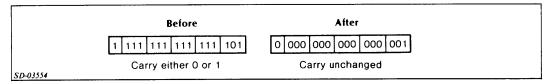
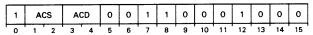


Figure 10.1 ADI example

AND with Complemented Source

ANC acs, acd



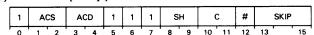
ANDs the contents of an accumulator with the logical complement of another accumulator.

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
0	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]

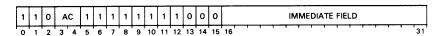
The processor can skip the next instruction if the condition test is true. The following list gives the test conditions, bits 13 to 15, and the operation.

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

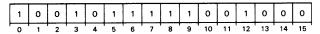


ANDs the contents of an accumulator with the contents of a 16-bit number contained in the instruction.

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.

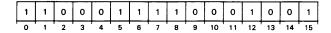
The 31-bit effective address generated by this instruction is constrained to be within the first 64 Kbytes of the current segment.

Words are moved in consecutive, ascending order according to their addresses. The next address after 77777₈ 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.

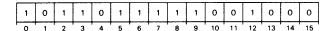
Before executing the **BKPT** instruction, you must first store, in memory, the one-word opcode from the location that the **BKPT** instruction will occupy. Then, store the **BKPT** instruction in that one-word location.

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).

Carry and overflow are indeterminate as a result of executing the **BKPT** instruction.

Block Move BLM



Moves memory words from one location to another.

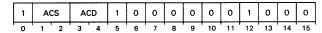
The *Block Move* instruction is the same as the *Block Add and Move* instruction in all respects except that no addition is performed and AC0 is not used. Carry remains unchanged and *overflow* is 0.

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.

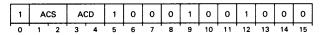
Forms a 32-bit bit pointer from the contents of bits 16-31 of both ACS and ACD. Bits 16-31 of ACS contain the high-order 16 bits and bits 16-31 of ACD contain the low-order 16 bits of the bit pointer. If ACS and ACD are specified as the same accumulator, the instruction treats the accumulator contents 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.

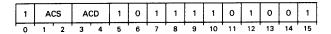
Forms a 32-bit bit pointer from the contents of bits 16-31 of both ACS and ACD. Bits 16-31 of ACS contains the high-order 16 bits and bits 16-31 of ACD contains the low-order 16 bits of the bit pointer. If ACS and ACD are specified as the same accumulator, the instruction treats the accumulator contents 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.

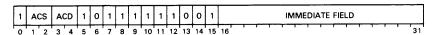
Bits 17–19 specify the I/O channel to use. I/O channel numbers range from 0–7.

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.

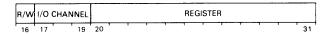
Command I/O Immediate

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).

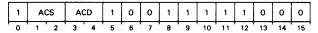
If ACS and ACD are different, then the logical OR of the immediate field and bits 16-31 of ACS is the command to be issued on the I/O channel bus.

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 16-31 of ACD. The limit value H is contained in the word following L. Bits 0-15 of ACD are ignored.

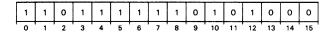
The 31-bit effective address generated by this instruction is constrained to be within the first 64 Kbytes of the current segment.

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 sequential word is 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 0-255₁₀. 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 2string 1 = string 20 string 1 > string 2

The strings may overlap in any way. Overlap will not effect the results of the comparison.

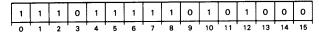
Upon completion, bits 16-31 of AC0 contain the number of bytes left to compare in string 2. AC1 contains the return code as shown in the list above. Bits 16-31 of AC2 contain a byte pointer either to the failing byte in string 2 (if an inequality were found) or to the byte following string 2 (if string 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 ACO 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.

١

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 0-255₁₀) 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 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.

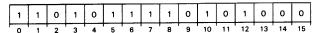
The 31-bit effective address generated by this instruction is constrained to be within the first 64 Kbytes of the current segment.

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 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 ACO contains the number 0 at the beginning of this instruction, no bytes are fetched and none are stored. If ACl is 0 at the beginning of this instruction, the destination field is filled with space characters.

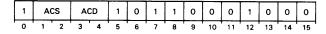
The original contents of AC2 and AC3 must be valid pointers to some area in the user's address space. If they are invalid, a protection fault occurs, even if no bytes are to be moved. AC1 contains the code 2.

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
0	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 10002 or 10012, other instructions use the bit combinations.

[skip]

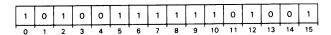
The processor can skip the next instruction if the condition test is true. The following list gives the test conditions, bits 13 to 15, and the operation.

Bits 13-15	Operation
0 0 0	Never skip
0 0 1	Always skip
0 1 0	Skip if CARRY is 0
0 1 1	Skip if CARRY is not 0
1 0 0	Skip if the result is 0
1 0 1	Skip if the result is not 0
1 1 0	Skip if either CARRY or the result is 0
1 1 1	Skip if both CARRY and the result are not 0
	13-15 0 0 0 0 0 1 0 1 0 0 1 1 1 0 0 1 0 1 1 1 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.

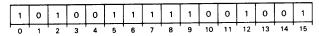
Complement Carry

CRYTC



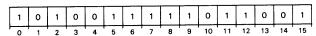
Complements the value of carry. Overflow is 0.

Set Carry to One CRYTO



Unconditionally sets the value of carry to 1. Overflow is 0.

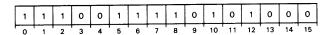
Set Carry to Zero CRYTZ



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 0–255₁₀. 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.

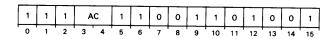
Code	Result
-1	Translated value of string 1 $<$ translated value of string 2
0	Translated value of string 1 = translated value of string 2
+1	Translated value of string 1 > translated value of string 2

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



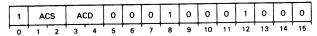
Converts a 32-bit integer to a 16-bit integer.

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.)

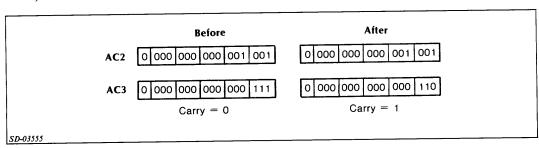
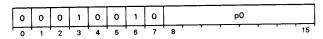


Figure 10.2 DAD example

Add to DI (edit subopcode)

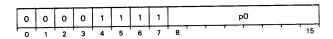
DADI p0



Adds the 8-bit two's complement integer specified by p0 to the Destination Indicator (DI).

Add to P Depending on S (edit subopcode)

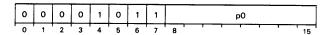
DAPS p0



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)

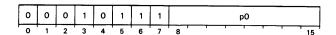
DAPT p0



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)

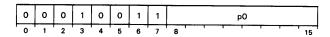
DAPU p0



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)

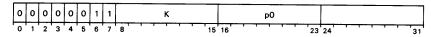
DASI p0



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

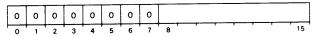


Decrements a word in the stack by one. If the decremented value of the word is nonzero, the instruction adds the 8-bit two's complement integer specified by p0 to the opcode Pointer (P). Before the add is performed, P is pointing to the byte containing the **DDTK** opcode.

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**

1	1	1	0	0	1	1	1	1	1	0	0	1	0	0	1
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

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.

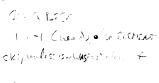
When all of the required pages are resident, the instruction then attempts to dequeue the data element.

- 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.

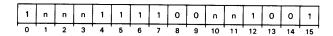
The dequeue operation is not interruptible. The entire operation finishes before any interrupts can be enabled. The contents of AC0, AC1, AC2, and AC3 remain unchanged. Carry is unchanged and overflow is 0.



1 - Demin 7

Detected Error

DERR nn



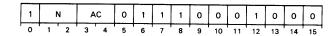
The DERR instruction pushes onto the wide stack the PC of the DERR instruction and a 5-bit code. The instruction then jumps to a user-supplied error (or trap) handler. The processor zero extends the 5-bit error code to 32 bits before pushing it on the stack.

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.

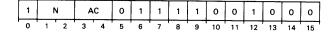
NOTES: If AC is specified as AC3, then AC+1 is AC0.

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.

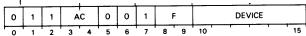
NOTES: If AC is specified as AC3, then AC+1 is AC0.

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



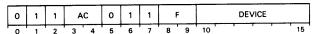
Transfers data from the A buffer of an I/O device to bits 16-31 of an accumulator.

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



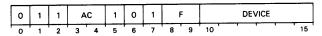
Transfers data from the B buffer of an I/O device to bits 16-31 of an accumulator.

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.

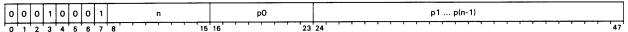
Places the contents of the C input buffer in the specified device in bits 16-31 of the specified accumulator. After the data transfer, sets the Busy and Done flags according to the specified F.

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)

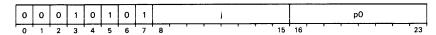
DICI n,p0[,p1,...,p(n-1)]



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)

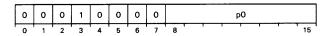
DIMC j,p0



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)

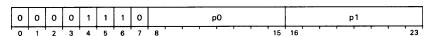
DINC p0



Inserts the character specified by p0 in the destination field at the position specified by DI. Increments DI by one.

Insert Sign (edit subopcode)

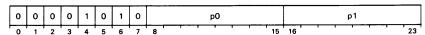
DINS *p0,p1*



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.

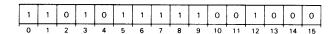
Insert Character Suppress (edit subopcode)

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.

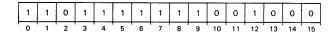
Divides the unsigned, 32-bit number contained in bits 16-31 of AC0 and bits 16-31 of AC1 by the unsigned, 16-bit number in bits 16-31 of AC2. The quotient and remainder are unsigned, 16-bit numbers and are placed in bits 16-31 of AC1 and AC0, respectively. Carry is set to 0. The contents of AC2 remain 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



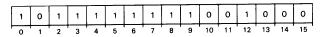
Divides the signed 32-bit integer in bits 16-31 of two accumulators by the signed contents of a third accumulator. The quotient and remainder each occupy one accumulator.

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.

DIVX



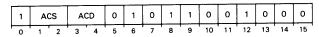
Extends the sign of one accumulator into a second accumulator and performs a Signed Divide on the result.

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.

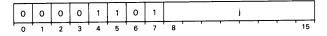
AC3+1 is AC0. The number of bits shifted is equal to the magnitude of the number in bits 24-31 of ACS. Bits shifted out are lost and the vacated bit positions are filled with zeroes. Carry and the contents of ACS remain unchanged. *Overflow* is 0.

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 31₁₀, bits 16–31 of ACD are set to 0. Carry and the contents of ACS remain unchanged.

Move Alphabetics (edit subopcode)

DMVA j



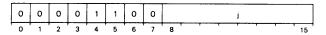
Moves j characters from the source field (beginning at the position specified by SI) to the destination field (beginning at the position specified by DI). Increases both SI and DI by j. Sets T to 1.

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

1



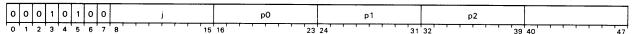
For EDIT, the DMVC instruction increments SI if the source data type is 3 and j>0. The instruction then moves j characters from the source field beginning at the position specified by SI to the destination field beginning at the position specified by DI. Increases SI and DI by j. Sets T to 1.

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.

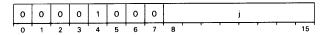
*

*

*

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.

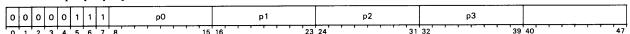
The DMVN instruction increases DI by j and sets T to 1. DMVN compares j to the remaining number of source characters to move and increases SI by the smaller of the two values if the source data type is 3, if j>0, and if SI points to the sign of the source number.

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.

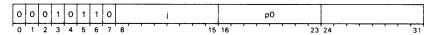
If the digit is a zero or space and S is 0, then the instruction places p0 in the destination field. If the digit is a zero or space and S is 1, then the instruction places p1 in the destination field. If the digit is a nonzero and S is 0, the instruction adds p2 to the digit and places the result in 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.

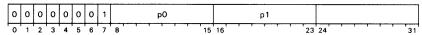
If the source data type is 2, the state of the SI is undefined after the least significant digit has been processed.

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)

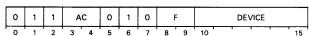
DNDF p0,p1



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



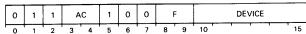
Transfers data from bits 16-31 of an accumulator to the A buffer of an I/O device.

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



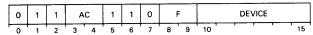
Transfers data from bits 16-31 of an accumulator to the B buffer of an I/O device.

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.

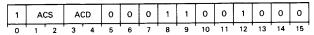
Places the contents of bits 16-31 of the specified accumulator in the C output buffer of the specified device. After the data transfer, sets the BUSY and DONE flags according to the function specified by F. The contents of the specified accumulator remain unchanged.

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.)

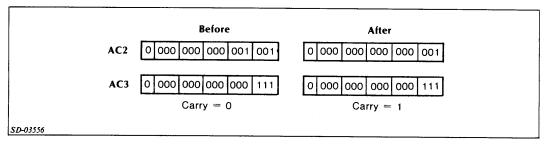
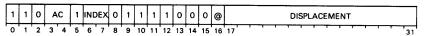


Figure 10.3 DSB example

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.

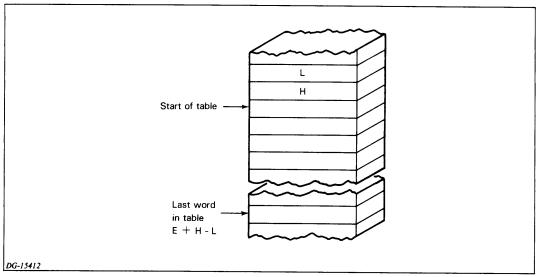


Figure 10.4 DSPA dispatch table structure

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 177777₈, sequential operation continues with the instruction immediately after the Dispatch instruction. If the fetched word is not equal to 1777778, 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.

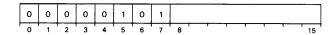
The 31-bit effective address generated by this instruction is constrained to be within the first 64 Kbytes of the current segment.

Carry is unchanged and overflow is 0.

Set S to One (edit subopcode)

DSSO

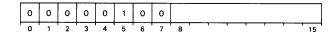
*



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

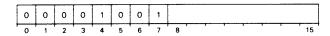


For EDIT, DSTK stores the byte specified by p0 in bits 8-15 of a word in the narrow stack. Sets bits 0-7 of the word that receives p0 to 0. If the 8-bit two's complement integer specified by k is negative, the instruction addresses the word receiving p0 by narrow stack pointer +1+k. If k is positive, 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)

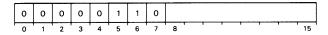
DSTO



Sets the significance Trigger (T) to 1.

Set T to Zero (edit subopcode)

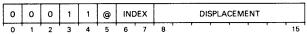
DSTZ



Sets the significance Trigger (T) to 0.

Decrement and Skip if Zero

DSZ [@]displacement[,index]



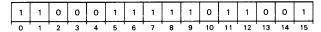
Decrements the addressed word, then skips if the decremented value is zero.

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

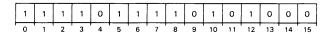
ECLID



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 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 1777778 in ACO.

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

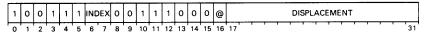
```
stack\ pointer + 1 + j.
```

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.

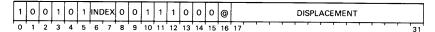
Computes the effective address, E. Decrements by one the contents of the location addressed by E and writes the result back into that location. If the updated value of the word is zero, the instruction skips the next sequential word.

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.

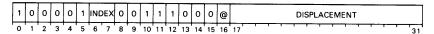
Computes the effective address, E. Increments by one the contents of the location specified by E and writes the new value back into memory at the same address. If the updated value of the location is zero, the instruction skips the next sequential word.

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.

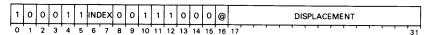
Computes the effective address, E, and places it in the program counter. Sequential operation continues with the word addressed by the updated value of the program counter.

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.

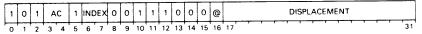
The 31-bit effective address generated by this instruction is constrained to be within the first 64 Kbytes of the current segment.

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.

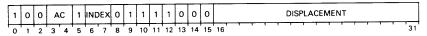
Calculates the effective address, E. Places the contents of the location addressed by E in bits 16-31 of the specified accumulator. The contents of the location addressed by E remain unchanged.

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.

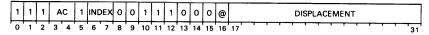
The argument *index* selects the source of the index value. It may have values in the range of 0-3. The following list gives the meaning of each value.

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

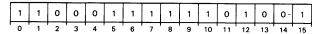
ELEF ac,[@]displacement[,index]



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.

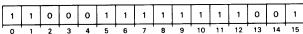
When all of the required pages are resident, the instruction then enqueues the data element addressed by AC2 before the data element addressed by AC1. If the new data element becomes the head of the queue, the instruction updates the queue descriptor appropriately. The next sequential word is skipped.

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 $AC1 \neq -1$, 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.

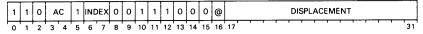
The instruction checks all reads and writes of links in data elements and queue descriptors against the current ring. Ring numbers of the link addresses must be greater than or equal to the current ring.

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.

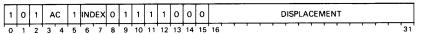
Places contents of bits 16–31 of the specified accumulator in the word addressed by the effective address, E. The previous contents of the location addressed by E are lost. The contents of the specified accumulator and carry remain unchanged.

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.

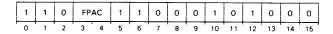
The argument index selects the source of the index value. It may have values in the range of 0-3. the following list gives the meaning of each value.

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

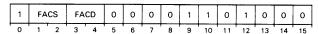


Sets the sign bit of FPAC to 0.

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

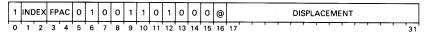


Adds the 64-bit floating-point number in FACS to the 64-bit floating-point number in FACD.

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]



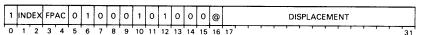
Adds the 64-bit floating-point number in the source location to the 64-bit floating-point number in FPAC and places the normalized result in FPAC.

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

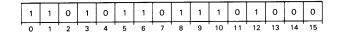
1	FACS	FACD	0	0	0	0	0	1	0	1	0	0	0
	1 2	3 4	5	6	7	8	9	10	11	12	13	14	15

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 Nflags in the floating-point status register to reflect the new contents of FACD.

Clear Errors

FCLE



Sets bits 0-4 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

1	FA	cs	FA	CD	1	1	1	0	0	1	0	1	0	0	0	
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	'

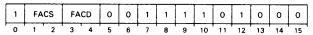
Compares two 64-bit floating-point numbers and sets the Z and N flags in the floating-point status register accordingly.

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< td=""></facd<>

Divide Double (FPAC by FPAC)

FDD facs, facd

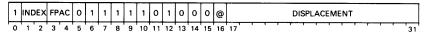


Divides the floating-point number in FACD by the floating-point number in FACS and places the normalized result in FACD.

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]



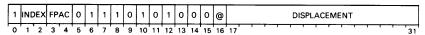
Divides the 64-bit floating-point number in FPAC by the 64-bit floating-point number in the source location and places the normalized result in FPAC.

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



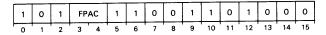
Divides the 32-bit floating-point number in FACD by the 32-bit floating-point number in FACS and places the normalized result in FPAC.

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 ACO. 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

1	Α	C	FP.	AC	1	0	1	1	0	1	0	1	0	0	0
-	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

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.

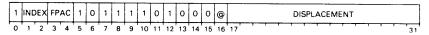
If the integer portion of the number contained in FPAC is less than -32,768 or greater than +32,767, the instruction sets MOF in the FPSR to 1. Takes the absolute value of the integer portion of the number contained in the FPAC. Takes the 15 least significant bits of the absolute value and appends a 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.

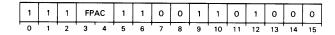
If the integer portion is within the range of -2.147.483.648 to +2.147.483.647 inclusive, the instruction places the 32-bit, two's complement representation of the integer portion of the number contained in the FPAC in the memory locations specified by E.

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.

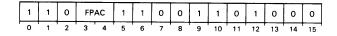
Shifts the mantissa contained in FPAC right one bit position. Fills the vacated bit position with a zero and places the bit shifted out in the guard digit. Normalizes the number and places the result in FPAC. Updates the Z and N flags in the floating-point status register to reflect the new contents 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.

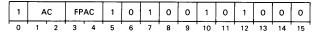
Zeros the fractional portion (if any) of the number contained in the specified FPAC. Normalizes the result. Updates the Z and N flags in the floating-point status register to reflect the new contents of the specified FPAC.

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

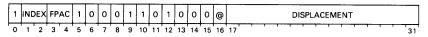


Converts a two's complement number in the range of -32,768 to +32,767 inclusive to floating-point format.

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.

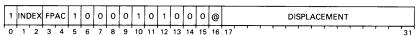
Computes the effective address, E. Fetches the double-precision floating-point number at the address specified by E and places it in FPAC. Updates the Z and N flags in the FPSR to reflect the new contents of FPAC.

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.

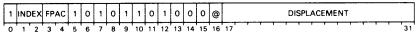
Computes the effective address, E. Fetches the single-precision floating-point number at the address specified by E. Places the number in the high-order bits of FPAC. Sets the low-order 32 bits of FPAC to 0. Updates the Z and N flags in the floating-point status register to reflect the new contents of 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.

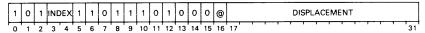
Computes the effective address, E. Converts the 32-bit, signed, two's complement number addressed by E to a double-precision floating-point number. Places the result in the specified FPAC. Updates the Z and N flags in the floating-point status register to reflect the new contents of the FPAC.

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

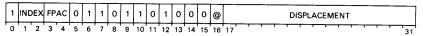
1	FA	cs	FA	CD	0 0	0	1	0	1	1	0	1	0	0	0
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

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]



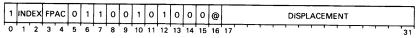
Multiplies the 64-bit floating-point number in the source location by the 64-bit floating-point number in FPAC and places the normalized result in FPAC.

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]



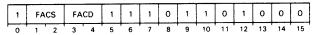
Multiplies the 32-bit floating-point number in the source location by the 32-bit floating-point number in bits 0-31 of FPAC and places the normalized result in FPAC.

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



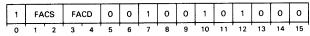
Moves the contents of one 64-bit FPAC to another 64-bit FPAC.

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

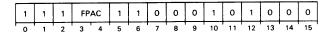


Multiplies the 32-bit floating-point number in bits 0-31 of FACS by the 32-bit floating-point number in bits 0-31 of FACD.

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



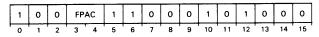
Inverts the sign bit of the FPAC.

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

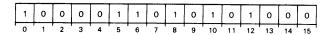


Normalizes the floating-point number in the 64-bit FPAC.

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.

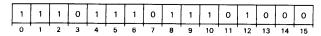
If an exponent underflow occurs, sets the *UNF* flag in the floating-point status register. In this case, the mantissa and the sign of the number in FPAC are correct, but the exponent is 128 too large.

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
- 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

This instruction does not initiate a floating-point trap under any conditions of the FPSR.

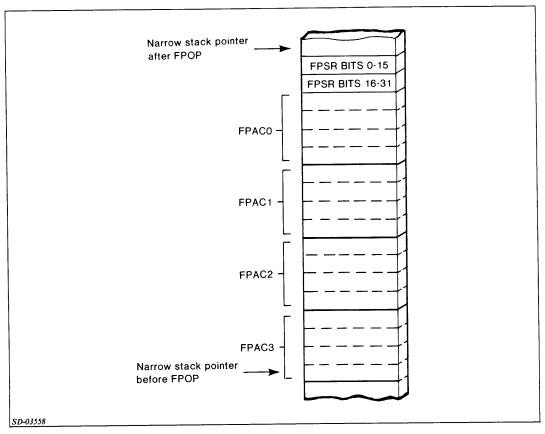
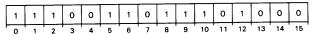


Figure 10.5 Narrow stack, 18-word block

Push Floating-Point State FPSH



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.

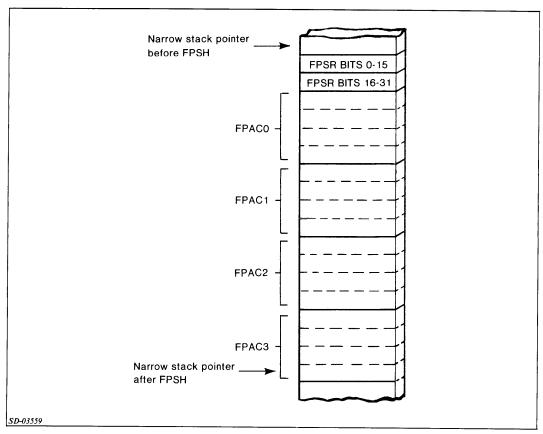


Figure 10.6 Narrow stack, 18-word floating-point return block

Floating-Point Round Double to Single

FRDS facs, facd

i	1	FA	cs	FA	CD	1	0	0	1	1	0	1	1	0	0	0
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

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 7FFFFFFF16 inclusive. The result mantissa is equal to the unrounded mantissa without change.
- 80000000₁₆. The result mantissa is formed by adding the least significant bit of the unrounded mantissa to the unrounded mantissa.
- 80000001₁₆ to FFFFFFFF₁₆ inclusive. The result mantissa is equal to the unrounded mantissa plus 1.

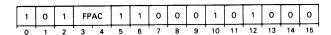
Algorithm is similar to unbiased rounding except that it uses eight rounding digits instead of two guard digits.

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

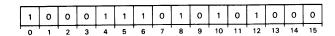


Places the high-order 16 bits of FPAC in bits 16-31 of AC0. FPAC and the Z and N flags in the floating-point status register remain unchanged.

NOTE: This instruction moves unnormalized data without change.

Skip Always

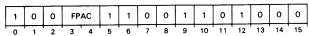
FSA



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.

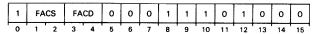
If D is negative, the instruction shifts the mantissa of the number contained in FPAC to the left by D digits. Sets the MOF flag in the floating-point status register.

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

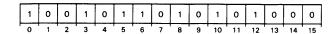


Subtracts the floating-point number in one FPAC from the floating-point number in another FPAC.

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

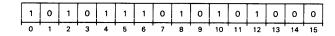
FSEQ



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

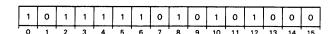
FSGE



Skips the next sequential word if the N flag of the floating-point status register is 0.

Skip on Greater than Zero

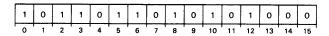
FSGT



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

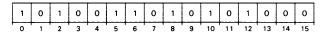
FSLE



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

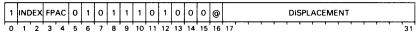
FSLT



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]



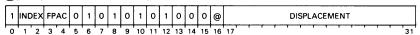
Subtracts the 64-bit floating-point number in the source location from the 64-bit floating-point number in FPAC and places the normalized result in FPAC.

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]



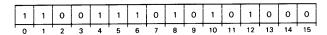
Subtracts the 32-bit floating-point number in the source location from the 32-bit floating-point number in bits 0-31 of FPAC and places the normalized result in FPAC.

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 Zero Divide

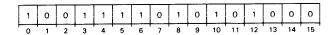
FSND



Skips the next sequential word if the DVZ (divide by zero) flag of the floating-point status register is 0.

Skip on Nonzero

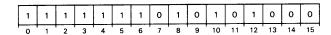
FSNE



Skips the next sequential word if the Z flag of the floating-point status register is 0.

Skip on No Error

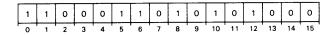
FSNER



Skips the next sequential word if bits 1-4 of the floating-point status register are all 0.

Skip on No Mantissa Overflow

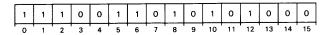
FSNM



Skips the next sequential word if the MOF (mantissa overflow) flag of the floating-point status register is 0.

Skip on No Overflow

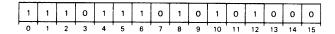
FSNO



Skips the next sequential word if the OVF (overflow) flag of the floating-point status register is 0.

Skip on No Overflow and No Zero Divide

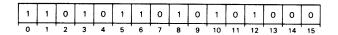
FSNOD



Skips the next sequential word if both the OVF flag and the DVZ flag of the floating-point status register are 0.

Skip on No Underflow

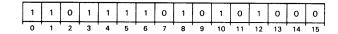
FSNU



Skips the next sequential word if the *UNF* (underflow) flag of the floating-point status register is 0.

Skip on No Underflow and No Zero Divide

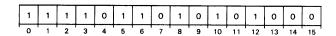
FSNUD



Skips the next sequential word if both the *UNF* flag and the *DVZ* flag of the floating-point status register are 0.

Skip on No Underflow and No Overflow

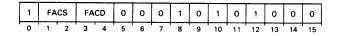
FSNUO



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

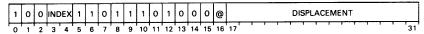


Subtracts the floating-point number in one FPAC from the floating-point number in another FPAC.

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.

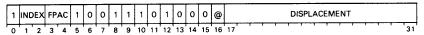
- Stores bits 0-15 of the FPSR in the first memory word.
- If ANY is 0, the contents of the second memory word are undefined.
- If ANY is 1, the instruction stores bits 48-63 of the FPSR into the second memory word.

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.

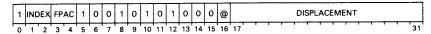
Computes the effective address, E. Places the floating-point number contained in FPAC in memory beginning at the location addressed by E. Destroys the previous contents of the addressed memory location. The contents of FPAC and the condition codes in the FPSR remain unchanged.

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.

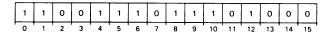
Computes the effective address E. Places the 32 high-order bits of FPAC in memory beginning at the location addressed by E. Destroys the previous contents of the addressed memory location. The contents of FPAC and the condition codes in the FPSR remain unchanged.

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

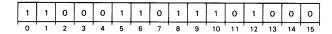


Sets the Trap Enable (TE) bit of the FPSR to 0.

NOTE: The I/O RESET instruction will also set this bit to 0.

Trap Enable

FTE



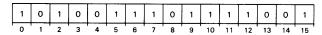
Sets the Trap Enable (TE) bit of the FPSR to 1.

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.

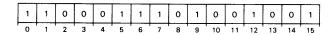
When a floating-point fault occurs and TE is 1, the processor sets TE to 0 before transferring control to the floating-point error handler. TE should be set to 1 before resuming normal processing.

Fixed-Point Trap Disable FXTD



Unconditionally sets the *OVK* flag to zero. This disables fixed-point overflow traps. Carry is unchanged.

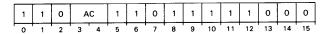
Fixed-Point Trap Enable FXTE



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.

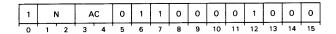
The signed, 16-bit two's complement number contained in bits 16-31 of the specified accumulator is divided by two and rounded toward zero. The result is placed in bits 16-31 of the specified accumulator.

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.

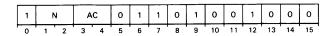
Shifts the contents of bits 16-31 of the specified accumulator left a number of hex digits depending upon the immediate field N. The number of digits shifted is equal to N+1. Bits shifted out are lost, and the vacated bit positions are filled with zeroes. If N is equal to 3, then bits 16-31 of the specified 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.

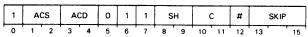
Shifts the contents of bits 16-31 of the specified accumulator right a number of hex digits depending upon the immediate field, N. The number of digits shifted is equal to N+1. Bits shifted out are lost and the vacated bit positions are filled with zeroes. If N is equal to 3, then bits 16-31 of the specified 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 177777₈, 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 10-1		Operation			
omitted	0	0	Leave CARRY unchanged			
Z	0	1	Initialize CARRY to 0			
0	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	10	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 10002 or 10012, other instructions use the bit combinations.

[skip]

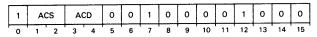
The processor can skip the next instruction if the condition test is true. The following list gives the test conditions, bits 13 to 15, and the operation.

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



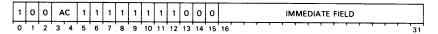
Inclusively ORs the contents of two accumulators.

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



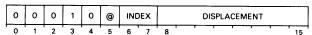
Inclusively ORs the contents of an accumulator with the contents of a 16-bit number in the instructions.

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]



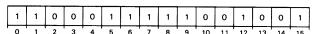
Increments the addressed word, then skips if the incremented value is zero.

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



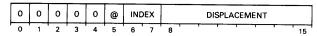
Increments the double word addressed by the wide stack pointer and skips the next 16-bit word if the incremented value is zero.

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.

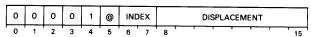
Computes the effective address, E, and places it in the program counter. Sequential operation continues with the word addressed by the updated value of the program counter.

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.

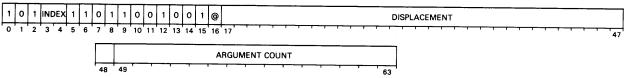
The 31-bit effective address generated by this instruction is constrained to be within the first 64 Kbytes of the current segment.

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]]



Evaluates the address of a subroutine call.

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.

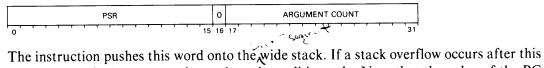
If the target address specifies an inward ring call, then the instruction assumes the target address has the following format:

х	NEW RING	UNUSED	0	GATE				
0	1 3	4	5 16	17 31				

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.

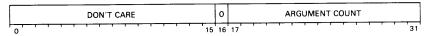
If the specified gate is legal, or if the target address specifies an intra-ring crossing, the instruction loads the contents of the PC, plus four, into AC3. The contents of AC3 always reference the current ring.

If bit 0 of the argument count is 0, the instruction creates a word with the following format:



The instruction pushes this word onto the wide stack. If a stack overflow occurs after this push, a stack fault occurs and no subroutine call is made. Note that the value of the PC in the return block is undefined.

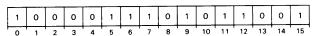
If bit 0 of the argument count is 1, then the instruction assumes the top word of the wide stack has the following format:



The instruction modifies this word to include the correct settings of the PSR.

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.

Load CPU Identification LCPID

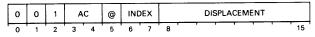


Loads a double word into ACO. Carry is unchanged and overflow is 0.

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.

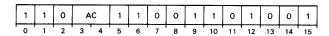
Bits 0-15 are undefined.

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

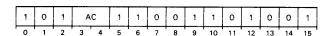


Loads the specified accumulator with the contents of WSB.

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

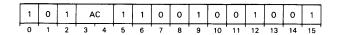


Loads the specified accumulator with the contents of WSL.

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

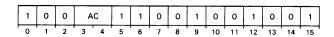


Loads the specified accumulator with the contents of WSP.

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

	1	A	cs	A	CD	1	0	1	1	1	0	0	1	0	0	0
•	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

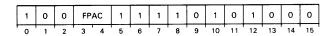
Moves a copy of the contents of a memory byte (as addressed by a byte pointer in one accumulator) into the second accumulator.

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.

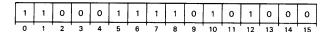
Upon successful completion, the instruction leaves accumulators AC0 and AC1 unchanged. AC2 contains the original contents of AC3. AC3 points to the first byte following the integer field. Carry remains unchanged and *overflow* is 0.

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 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 $Wide\ Dispatch$ 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 *Wide Dispatch* instruction. If the fetched double word is not equal to 3777777777_8 , the instruction adds the double word to its address 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.

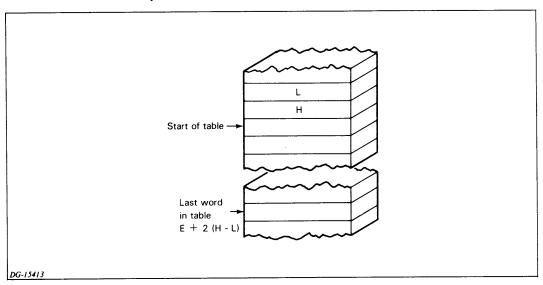
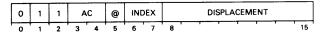


Figure 10.7 LDSP dispatch table structure

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.

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.

Carry is unchanged and overflow is 0.

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]



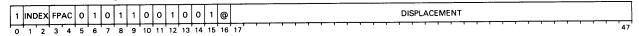
Moves four words out of memory and into a specified FPAC.

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 33-63 of (FPPC) the operand in bits 33-63 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

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]

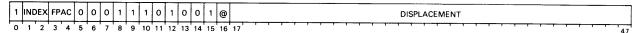


Subtracts the 64-bit floating point number in the source location from the 64-bit floating point number in FPAC and places the normalized result in FPAC.

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.
- 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 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]



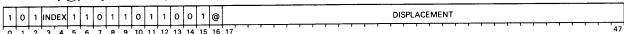
Stores the contents of a specified FPAC into a memory location.

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 a byte pointer 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.

Load Effective Byte Address (Long Displacement)

LLEFB ac, displacement [, index]

*

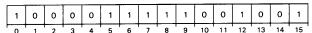
*



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 ACO.

AC1 contains a pageframe number in bits 13-31.

The bits are loaded into ACO, 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]

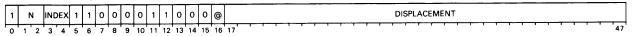


Adds an integer contained in a memory location to an integer contained in an accumulator.

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]

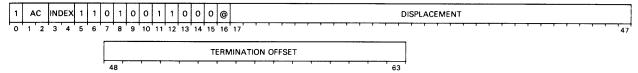


Divides an integer contained in an accumulator by an integer in memory.

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]



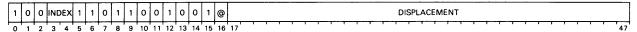
Decrements the contents of a location and skips the next word if the decremented value is zero.

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]



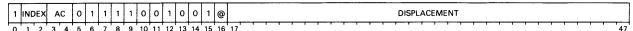
Increments the contents of a location and skips the next word if the incremented value is zero.

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]

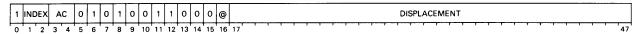


Sign extends and loads the contents of a memory location into an accumulator.

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]

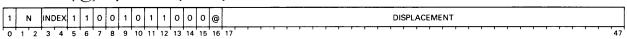


Multiplies an integer in memory by an integer in an accumulator.

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]



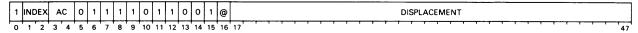
Subtracts an integer in the range of 1 to 4 from an integer contained in a 16-bit memory location.

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]

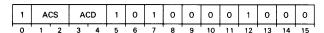


Subtracts an integer in memory from an integer in an accumulator.

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/

	1	0	1	INDEX	1	1	0	1	1	1	1	1	0	0	1	@	DISPLACEMENT
-	ο'	1 '	2	3 4	5	6	7	8	9	10	11	12	13	14	15	16	17

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.

Push Byte Address (Long Displacement)

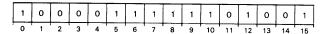
LPEFB displacement/,index/



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, 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, 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]

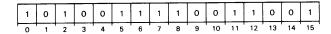
Г	1	0	INDEX	1	1	0	1	1	0	О	1	0	О	1	@	DISPLACEMENT
1	+	+-	3 4	Ļ	6	⊣	8	9	10	11	12	13	14	15	16	17 47

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

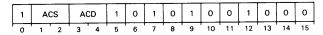


Loads the contents of the PSR into ACO.

Loads the contents of *OVK*, *OVR*, and *IRES* into bits 0, 1, and 2 of AC0, respectively. Fills the rest of AC0 with zeroes. The contents of the PSR remain unchanged. Carry is unchanged and *overflow* is 0.

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	SBRO	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 SBRO, 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.

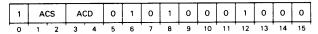
If SBR0 contains invalid information, then the processor disables the address translator and a protection fault occurs (code = 3 in AC1). This means that logical addresses are identical to physical addresses and the fault is processed in physical address space.

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.

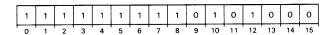
Shifts the contents of bits 16-31 of ACD either left or right depending on the number contained in bits 24-31 of ACS. The signed, 8-bit two's complement number contained in bits 24-31 of ACS determines the direction of the shift and the number of bits to be shifted. If the number in bits 24-31 of ACS 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]

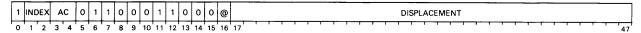


Stores the low-order byte of the specified accumulator in memory.

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, $\lceil (a) \rceil$ displacement $\lceil index \rceil$



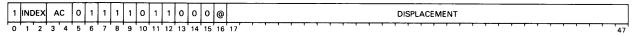
Adds an integer in the range of 1 to 4 to an integer contained in a 32-bit memory location.

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.

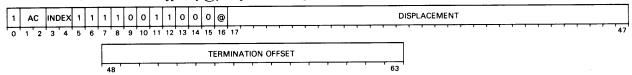
If the quotient is within the range of -2.147.483.648 to +2.147.483.647 inclusive, or if the memory word is zero, the instruction loads the quotient into the specified accumulator. Overflow is 0.

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.

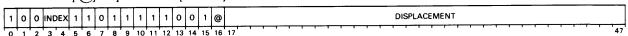
Increments a 32-bit memory location and compares it to the AC. If the memory location is greater than the AC, then a PC relative branch is made by adding the termination offset to PC+1. If the memory location is less than or equal to the AC, then the next instruction is executed. In either case, the 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]



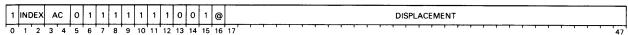
Increments the contents of a location and skips the next word if the incremented value is zero.

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]



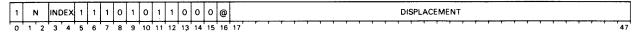
Multiplies an integer in an accumulator by an integer in memory.

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]

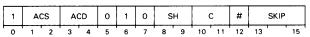


Subtracts an integer in memory from an integer in an accumulator.

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/	Bi1 10-		Operation
omitted	0	0	Leave CARRY unchanged
Z	0	1	Initialize CARRY to 0
0	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 10002 or 10012, other instructions use the bit combinations.

[skip]

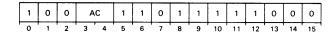
The processor can skip the next instruction if the condition test is true. The following list gives the test conditions, bits 13 to 15, and the operation.

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.

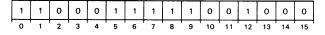
Adds the signed two's-complement number in bits 16-31 of the specified accumulator to the value of the stack pointer and places the result in location 40. The instruction then checks for overflow by comparing the result in location 40 with the value of the stack limit. If the result in location 40 is less than 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.

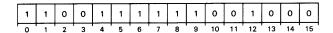
Multiplies the unsigned, 16-bit number in bits 16-31 of AC1 by the unsigned, 16-bit number in bits 16-31 of AC2 to yield an unsigned, 32-bit intermediate result. The unsigned, 16-bit number in bits 16-31 of AC0 is added to the intermediate result to produce the final result. The final result is an unsigned, 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.

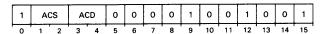
Multiplies the signed, 16-bit two's complement number in bits 16-31 of AC1 by the signed, 16-bit two's complement number in bits 16-31 of AC2 to yield a signed, 32-bit two's complement intermediate result. The signed, 16-bit two's complement number in bits 16-31 of AC0 is added to the intermediate result 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

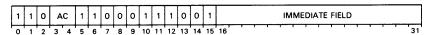


Adds two integers contained in accumulators.

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

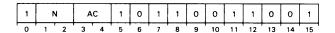


Adds an integer contained in an immediate field to an integer in an accumulator.

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

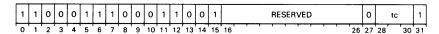


Adds an integer in the range of 1 to 4 to an integer contained in an accumulator.

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 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	000	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	100	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.

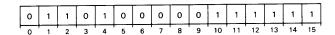
If the search fails, AC1 contains the effective address of the last data element searched, and the processor then executes the next instruction. The processor will honor interrupts between the time that it completes an unsuccessful search and executes the PC+2 instruction.

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

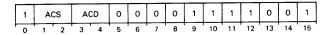


Loads CPU identification into the accumulators. Refer to the specific functional characteristic manual for the accumulator format. Carry is unchanged and overflow is 0.

NOTE: This instruction can be executed only with Lef mode and I/O protection disabled.

Narrow Divide

NDIV acs.acd

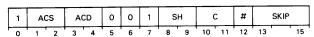


Divides an integer in an accumulator by an integer in another accumulator.

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/f 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

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/	Bit 10-		Operation
omitted	0	0	Leave CARRY unchanged
Z	0	1	Initialize CARRY to 0
0	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 10002 or 10012, other instructions use the bit combinations.

[skip]

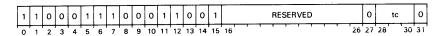
The processor can skip the next instruction if the condition test is true. The following list gives the test conditions, bits 13 to 15, and the operation.

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 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	000	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	100	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.

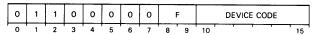
If the search fails, AC1 contains the effective address of the last data element searched, and the processor then executes the next instruction. The processor will honor interrupts between the time that it completes an unsuccessful search and executes the PC+2 instruction.

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



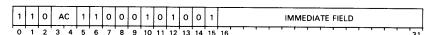
Used when a BUSY or DONE flag must be changed with no other operation taking place.

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

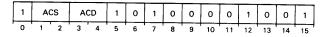
NMUL acs.acd

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



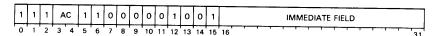
Negates an integer contained in an accumulator.

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 (100000₈) sets overflow to 1.

Narrow Skip on All Bits Set in Accumulator

NSALA i,ac

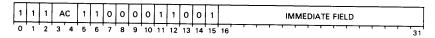


Logically ANDs the value in the immediate field with the complement of the contents of a memory word and skips depending on the result of the AND.

Performs a logical AND on the contents of the immediate field and the complement of the 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

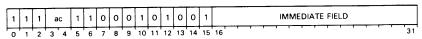


Logically ANDs the contents of an immediate field with the complement of the contents of a memory word and skips depending on the result.

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

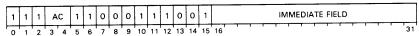


Logically ANDS the contents of an immediate field with the contents of an accumulator and skips depending on the result.

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

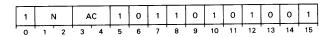


Logically ANDs the contents of an immediate field with the contents of a memory word and skips depending on the result.

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



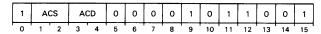
Subtracts a value in the range of 1 to 4 from the value contained in an accumulator.

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

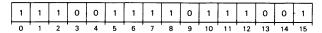
NSUB acs.acd



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 0000₁₆) 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.

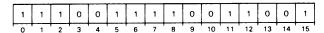
The instruction fetches the referenced bits of 16 consecutive pageframes, beginning with the pageframe specified by AC1. Exclusively ORs these 16 bits with the 16-bit word specified by AC2. Stores the result of the OR in the location specified by AC2. Resets the 16 referenced bits to 0, decrements AC0 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 ACI 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.

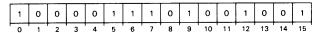
Purge the Address Translator PATU



Purges the entire address translator of all entries. Carry is unchanged and overflow is 0.

NOTE: 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.)

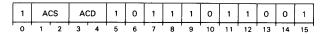
If the value popped off the stack and loaded into the PC does not reference the **BKPT** instruction, undefined results occur.

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

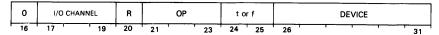
PIO acs, acd



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:



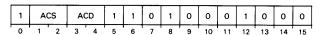
The instruction issues the command contained in ACS to the specified device. Bits 17–19 specify the I/O channel. Bits 21-23 (OP) contain the I/O operation code (bits 5-7 of an I/O instruction). Bits 8 and 9 (PULSE) contain the device flag command (bits 8 and 9 of an I/O instruction).

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 ACO 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 XOPO Extended operation or an I/O interrupt handler that does not use the stack change facility of the *Vector* instruction (VCT).

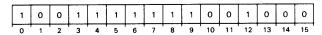
Five words are popped off the stack and placed in predetermined locations. The words popped and their destinations are as follows:

Destination
Bit 0 is loaded into carry
Bits 1-15 are loaded into the PC
AC3
AC2
AC1
ACO

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



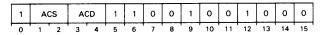
Pops the top word off the stack and places it in the program counter. Sequential operation continues with the word addressed by the updated value of the program counter.

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.

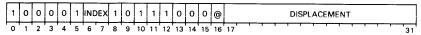
Bits 16-31 of the set of accumulators from ACS through ACD are pushed onto the stack. The contents of bits 16-31 of the accumulators are pushed in ascending order, starting with bits 16-31 of the AC specified by ACS and continuing up through bits

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.

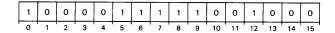
Pushes the address of the next sequential instruction onto the stack, computes the effective address, E, and places it in the program counter. Sequential operation continues with the word addressed by the updated value of the program counter.

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

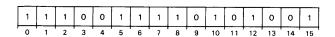


Pushes the address of this instruction plus two onto the narrow stack.

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 the number of groups of 16 referenced bits to reset. A count of 0 (or any negative count except 8000 0000₁₆) means that the instruction resets 16 pageframes.

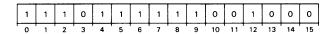
The instruction sets to 0 the referenced bits of 16 contiguous pageframes, starting with the pageframe specified by the contents of AC1. Decrements the contents of AC0 by 1 and increments the contents of AC1 by 16.

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 ACO 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.

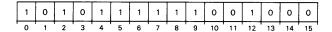
Carry remains unchanged and overflow is 0.

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 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

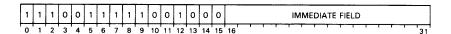
After popping the return block, the *Return* instruction loads the decremented value of the frame pointer into the stack pointer and the popped value of AC3 into the frame pointer.

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 ACO
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.

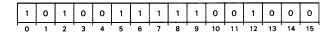
The Save instruction allocates a portion of the stack for use by the procedure which executed the Save. The value of the frame size, contained in the immediate field, determines the number of words in this stack area. This portion of the stack will not normally be accessed by push and pop operations, 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 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 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 ACO
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.

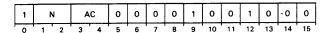
If an overflow condition exists, the Save Without Arguments instruction transfers control to the stack fault routine. The program counter in the fault return block contains the address of the Save Without Arguments instruction.

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 000003₈. After the instruction **SBI 4,2** is executed, bits 16-31 of AC2 contain 177777₈ and carry remains unchanged (see Figure 10.8.)

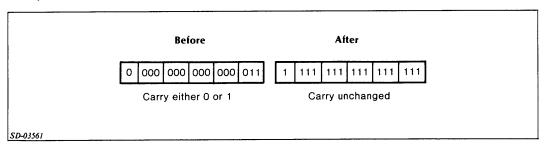
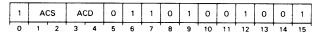


Figure 10.8 SBI example

Sign Extend

SEX acs, acd

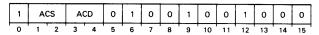


Sign extends the 16-bit integer in an accumulator to 32 bits.

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



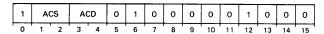
Compares two signed integers in two accumulators and skips if the first is greater than or equal to the second.

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



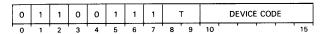
Compares two signed integers in two accumulators and skips if the first is greater than the second.

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

SKPt device



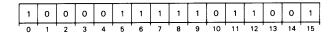
The SKP instruction tests the BUSY and DONE status flags of devices. If the test condition specified by t is true, the instruction skips the next sequential word.

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.

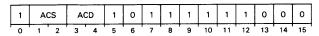
The instruction fetches the contents of the two least significant bits of AC0. Stores these values in the modified (bit 30) and referenced (bit 31) bits of the pageframe specified by AC1. Carry is unchanged and *overflow* is 0.

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.

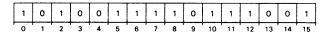
Forms a 32-bit bit pointer from the contents of bits 16-31 of both ACS and ACD. Bits 16-31 of ACS contain the high-order 16 bits and bits 16-31 of ACD contain the low-order 16 bits of the bit pointer. If ACS and ACD are specified as the same accumulator, the instruction treats the accumulator contents 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.

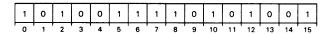
Skip on OVR Reset

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



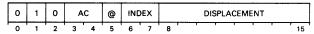
Stores the most significant bits of AC0 in the most significant bits of PSR.

The contents of AC0 remain unchanged. Carry is unchanged and overflow is 0.

Store Accumulator

*

STA ac,[@]displacement[,index]



Stores the contents of bits 16-31 of an accumulator into a memory location.

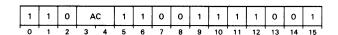
Places the contents of bits 16-31 of the specified accumulator in the word addressed by the effective address, E. The previous contents of the location addressed by E are lost.

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

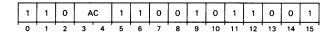


Stores the contents of an accumulator in WFP.

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

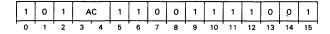


Stores the contents of an accumulator in WSB and updates the stack parameter in page zero of memory.

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

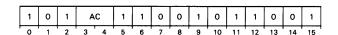


Stores the contents of an accumulator in WSL and updates the stack parameter in page zero of memory.

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

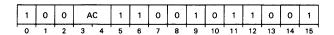


Stores the contents of an accumulator in WSP.

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

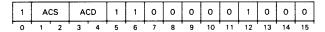


Stores the contents of an accumulator in the location addressed by WSP.

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.

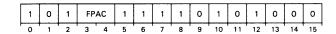
Places bits 24–31 of ACD in the byte addressed by the byte pointer contained in bits 16–31 of ACS.

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.

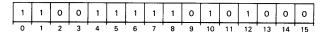
For data types 0 through 6, if the destination field cannot contain the entire number being stored, high-order digits are discarded until the number will fit into the destination. The remaining low-order digits are stored and carry is set to 1. For data type 7, the low-order digits are discarded.

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.

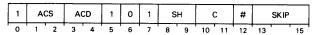
NOTES: If the number in the specified FPAC has any fractional part, the result of the instruction is undefined. Use the Intergerize instruction to clear any fractional part.

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]	Bit 10-		Operation
omitted	0	0	Leave CARRY unchanged
Z	0	1	Initialize CARRY to 0
0	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.

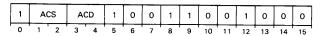
Forms a 32-bit bit pointer from the contents of bits 16-31 of both ACS and ACD. Bits 16-31 of ACS contain the high-order 16 bits and bits 16-31 of ACD contain the low-order 16 bits of the bit pointer. If ACS and ACD are specified as the same accumulator, the instruction treats the accumulator contents 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



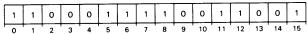
The two accumulators form a bit pointer. The instruction sets the addressed bit to 1. If the addressed bit was 0 before being set to 1, the instruction skips the next sequential word.

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: The processor, not the **VBP** instruction, checks for a valid access code (read, write, or execute.)

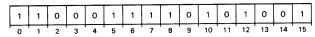
AC0 contains a 32-bit byte pointer. AC1 contains a segment number in bits 1-3; all other bits contain zeroes.

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: The processor, not the **VWP** instruction, checks for a valid access (read, write, or execute) and checks for more than 15 indirect addresses.

AC0 contains a 31-bit, indirectable word pointer. AC1 contains a segment number in bits 1-3; all other bits 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:

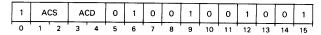
- 1. The segment number in AC0 is equal to or greater than the segment number in AC1.
- 2. The segment number in an indirect address is equal to or greater than the segment number in AC1 and the currently referenced segment.
- 3. If an indirection to a higher numbered segment is followed by another indirection, 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

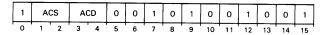


Adds the logical complement of the 32-bit integer specified by ACS to the 32-bit integer specified by ACD.

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

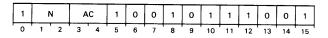


Adds the 32-bit integer in the immediate field to the 32-bit integer in the specified accumulator.

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



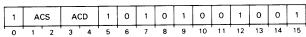
Adds the value specified by n+1 to the 32-bit integer in the specified accumulator.

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

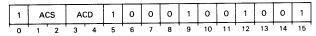


Logically ANDs the one's complement of the 32 bits specified by ACS with the 32 bits specified by ACD.

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

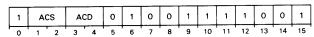


Logically ANDs the contents of the specified accumulator with the contents of the immediate field.

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.

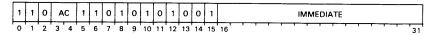
If the instruction is to shift the contents of ACD to the right, it truncates the contents one bit position for each shift.

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, *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.

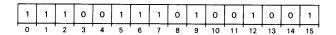
If the instruction is to shift the contents of AC to the right, it truncates the contents one bit position for each shift.

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, 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 decrements the count in AC1 by 1. If data movement is ascending, the instruction increments the source and destination addresses by 1 for each word moved. If data movement is descending, the instruction decrements the source and destination addresses 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.

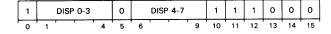
NOTES: Since this instruction may require a long time to execute, it is interruptible. When this instruction is interrupted, the processor saves the address of the WBLM instruction. This instruction updates addresses and word count after storing each word, so any interrupt service routine returning control 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. ACl will contain the value 4.

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

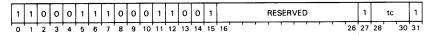


Adds a specified value to the program counter.

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 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	000	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.

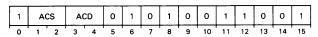
If the search fails, AC1 contains the effective address of the last data element searched, and the processor then executes the next instruction. The process will honor interrupts between the time that it completes an unsuccessful search and executes the PC+2 instruction.

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 word address.

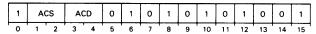
ACD contains a word offset and a bit pointer.

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 pointer.

Wide Set Bit to Zero

WBTZ acs,acd



Sets the specified bit to zero.

ACS contains a 31-bit word address.

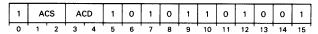
ACD contains a word offset and a bit pointer.

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 pointer.

Wide Compare to Limits

WCLM acs,acd



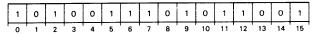
Compares a signed integer with two limit values and skips if the integer is between the limit values. The accumulators determine the location of the limit values.

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 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 0-255₁₀. 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.

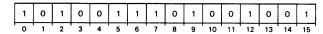
Upon completion, AC0 contains the number of bytes left to compare in string 2. AC1 contains the return code as shown in the list above. AC2 contains a byte pointer either to the failing byte in string 2 (if an inequality was found) or to the byte following string 2 (if string 2 was exhausted). AC3 contains 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 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

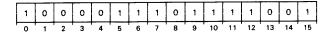
either to the delimiter or to the first byte following the source string. The value of carry is indeterminate and overflow is 0.

NOTES: The original contents of ACO, 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. ACI 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.

ACO 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, ACO 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, ACO 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 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, the instruction returns the value 0 in carry.

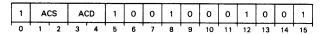
NOTES: If AC0 contains the number 0 at the beginning of this instruction, no bytes are fetched and none are stored. If AC1 is 0 at the beginning of this instruction, the destination field is filled with space characters; note that AC3 must still contain a valid byte pointer.

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 segment 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

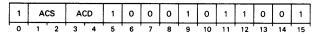


Counts and adds the number of ones in an accumulator to another accumulator.

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

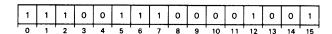
WCOM acs, acd



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 0-255₁₀) 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.

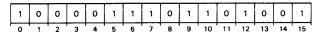
AC1 specifies the length of the string and the direction of processing. If the string is scanned in ascending order, AC1 contains the unsigned value of the number of bytes in the string. If the string is scanned in descending order, AC1 contains the two's complement of the number of bytes in the string.

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 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 ACO 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. ACI 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 0-255₁₀. 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

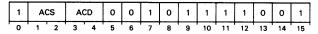
If the length of both string 1 and string 2 is zero, the compare option returns a 0 in AC1.

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 Divide

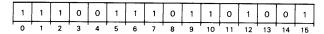
WDIV acs, acd



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



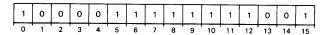
Divides an integer contained in AC0 and AC1 by an integer contained in AC2.

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.

Pop Context Block WDPOP

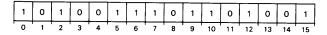


Restores the state of the machine to what it was at the time of the last page fault.

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 unchanged and overflow is 0.

NOTE: This is a privileged instruction.

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.

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

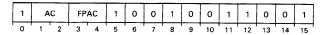
*

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.

Wide Fix from Floating-Point Accumulator

WFFAD ac, fpac



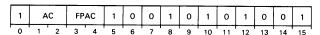
Converts the integer portion of the floating-point number contained in the specified FPAC to a 32-bit, signed, two's complement integer. Places the result in an accumulator.

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

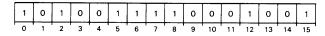


Converts the contents of a 32-bit accumulator to floating-point format and places the result in a specified FPAC.

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.

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.

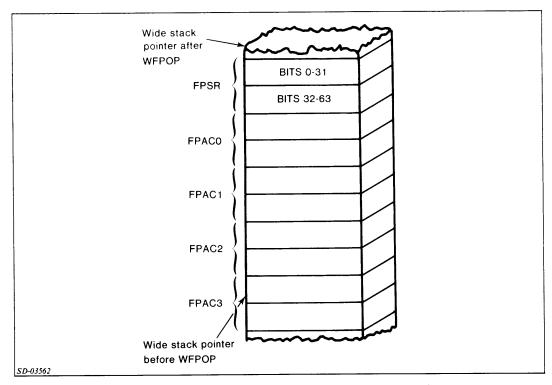


Figure 10.9 WFPOP 10 double-word

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 33-63 of the operand in bits 33-63 of the FPSR.

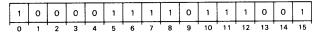
NOTE: This instruction moves unnormalized data without change.

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



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.

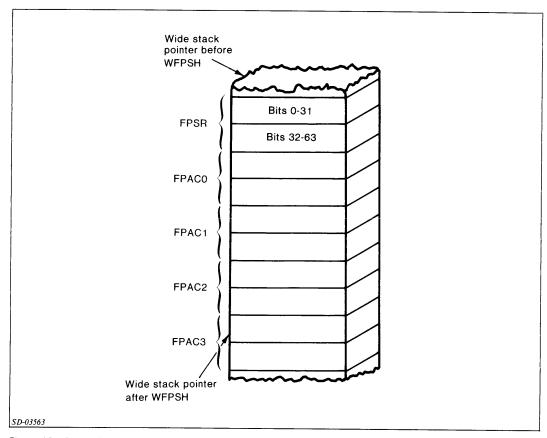


Figure 10.10 WFPSH 10 double-word block pushed

The instruction pushes the FPSR onto the stack as follows:

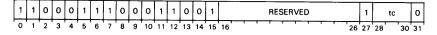
- Stores bits 0-15 of the FPSR in the first memory word.
- Sets bits 16-31 of the first memory double word and bit 0 of the second memory double word to 0.
- If ANY is 0, the contents of bits 1-31 of the second memory double word are undefined.
- If ANY is 1, the instruction stores bits 33–63 of the 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.

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 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	100	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

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 instructions. The processor will not honor interrupts between the time that it completes a successful search and executes the PC+4 instruction.

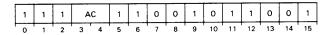
If the search fails, AC1 contains the effective address of the last data element searched, and the processor then executes the next instruction. The processor will honor interrupts between the time that it completes an unsuccessful search and executes the PC+2 instruction.

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.

Wide Halve

WHLV ac



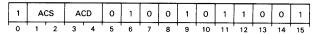
Divides the 32-bit contents of the specified accumulator by two and rounds the result toward zero.

The signed, 32-bit two's complement number contained in the specified accumulators 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

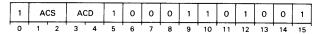


Increments an integer contained in an accumulator.

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

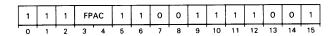


Loads a byte in memory into an accumulator.

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.

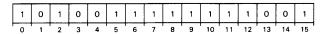
AC1 must contain the data-type indicator describing the integer.

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.

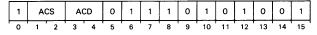
If bits 16–31 of AC1 all initially contain zeros, the instruction performs no operation.

NOTE: This is a privileged instruction.

Refer to the appropriate functional characteristics manual for more information.

Wide Locate Lead Bit

WLOB acs, acd

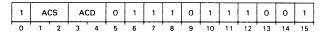


Counts the number of high-order zeroes in ACS.

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



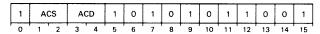
Counts the number of high-order zeroes in ACS.

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



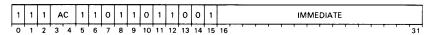
Shifts the 32-bit contents of ACD either left or right.

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.

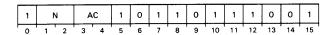
Bits 24-31 specify the number of bits to shift AC. If this number is positive (1 to 32_{10}), then the instruction shifts the contents of AC the appropriate number of bits to the left. If this number is negative (-1 to -32_{10}), then the instruction shifts the contents of AC the appropriate number of 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



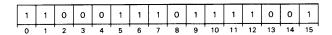
Shifts the contents of an accumulator left, as indicated by an immediate value.

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.

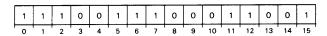
The instruction evaluates a decimal number in memory and returns in AC1 a code that classifies the number as zero or nonzero and identifies its sign. The meaning of the returned code is as follows:

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 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.

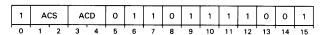
AC1 contains 32 bits that the processor exchanges with the 32 bits in memory when the result of the comparison is true. Upon completion, AC1 always contains the initial 32 bits of the double word addressed by AC2. (The processor pushes a fault return block and updates AC1 with an error code if a protection 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

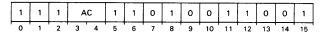


Moves a copy of the 32-bit contents of ACS into ACD.

The contents of ACS remain unchanged. Carry is unchanged and overflow is 0.

Wide Move Right

WMOVR ac



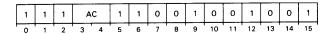
Converts a byte pointer to a word pointer.

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.

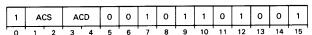
If no overflow occurs, the instruction checks the value of the accumulator. If the value is positive, the processor checks the temporary WSP against the stack limit for stack overflow; if negative, against the stack base for stack underflow. If underflow or overflow does not occur, the instruction places 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



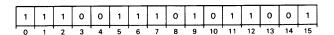
Multiplies two integers contained in accumulators.

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

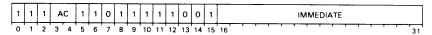


Multiplies two integers contained in accumulators.

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

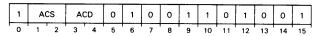


Adds an immediate value to an integer contained in an accumulator.

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

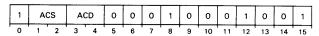


Negates the contents of an accumulator.

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.

Wide Pop Accumulators

WPOP acs, acd

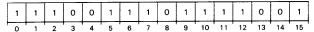


Pops up to four double words off the top of the wide stack and places them in the specified 32-bit accumulators.

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.

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	ACO
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.

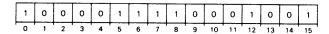
If the instruction specifies an intra-ring address, it pops the six-double-word block and saved frame area, then checks for stack underflow. If underflow has occurred, a stack underflow fault results. Note that the return block pushed as a result of the stack underflow will contain undefined information. 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) $- (2 \times (frame \ size))$

If argument count 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.

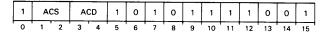
Wide Pop PC and Jump WPOPJ



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



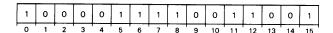
Pushes the contents of the specified 32-bit accumulators onto the top of the wide stack.

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
10	WFP	32 bits	
9	WSP	32 bits	
8	WSL	32 bits	
7	WSB	32 bits	
6	0,SFA,PSR	32 bits	< Stack fault address (bits 1-15)
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.

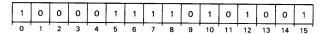
The instruction checks to see if the ring crossing specified is inward. If the crossing is inward, a protection fault occurs (code=8 in AC1).

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 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. 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.

If the ring crossing is outward, the instruction stores the stack management information held 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

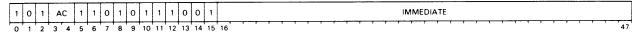


Logically ANDs an immediate value with the contents of an accumulator and skips depending on the result of the AND.

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

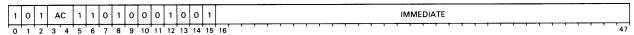


Logically ANDs an immediate value with the complement of a memory word and skips depending on the result of the AND.

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

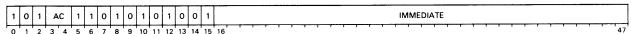


Logically ANDs an immediate value with the contents of an accumulator and skips depending on the result of the AND.

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

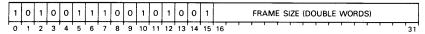


Logically ANDs an immediate value with the contents of a memory word and skips depending on the result of the AND.

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

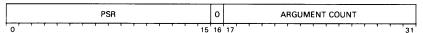


Pushes a return block onto the wide stack, resets OVK, and increments the wide stack pointer by frame size.

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	ACO
2	AC1
3	AC2
4	Previous WFP
5	Bit 0 equals CARRY Bits 1-31 equal AC3 bits 1-31 (or return PC value for XCALL and LCALL)

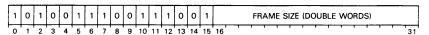
Note that the five words described above do not make up the entire return block. Either the LCALL or the XCALL instruction pushes the first double word of the return block onto the wide stack. This word has the following format:



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

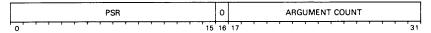


Pushes a return block onto the wide stack, resets WSP and WFP, sets OVK to 1, and increments the wide stack pointer by frame size.

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	ACO
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).

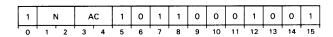
Note that the five double words described above do not make up the entire return block. Either the LCALL or the XCALL instruction pushes the first double word of the return block onto the wide stack. This word has the following format:



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



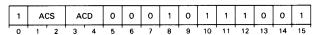
Subtracts an integer in the range 1 to 4 from an integer contained in an accumulator.

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



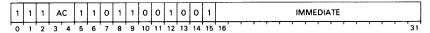
Compares one integer to another and skips if the two integers are equal.

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



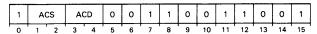
Compares one integer to another and skips if the first is equal to the second.

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



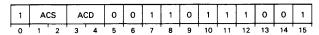
Compares one integer to another and skips if the first is greater than or equal to the second.

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



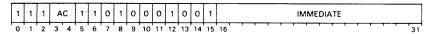
Compares one integer to another and skips if the first is greater than the second.

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



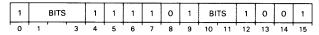
Compares one integer to another and skips if the first is greater than the second.

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

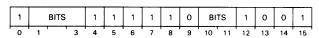


Tests a specified bit in AC0 and skips if the bit is 1.

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 ACO; 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 ACO remain unchanged. Carry is unchanged and overflow is 0.

Wide Skip on Bit Set to Zero

WSKBZ bit number

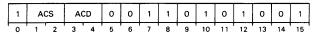


Tests a specified bit in AC0 and skips if the bit is 0.

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 ACO; 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 ACO remain unchanged. Carry is unchanged and *overflow* is 0.

Wide Signed Skip if Less than or Equal to

WSLE acs, acd



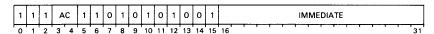
Compares one integer to another and skips if the first is less than or equal to the second.

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



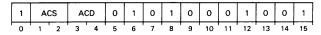
Compares one integer to another and skips if the first is less than or equal to the second.

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



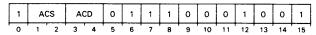
Compares one integer to another and skips if the first is less than the second.

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



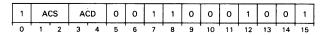
Tests the value of an addressed bit and skips if the bit is one.

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



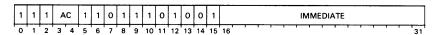
Compares one integer to another and skips if the two are not equal.

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.

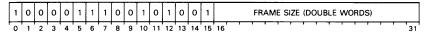
Sign extends the 16-bit immediate field. Compares this 32-bit number to the contents of the AC. If the contents of the AC are not equal to the contents of the immediate, then the next sequential word is skipped; otherwise, the next word is executed.

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:

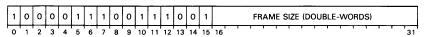
Double Word in Block	Contents
1	PSR, and zeroes in the unused bits
2	ACO
3	AC1
4	AC2
5	Previous WFP
6	Carry and AC3 1-31 (or return PC)

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:

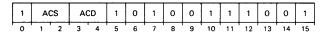
Double Word in Block	Contents
1	PSR, and zeroes in the unused bits
2	ACO
3	AC1
4	AC2
5	Previous WFP
6	Carry and AC3 1-31 (or return PC)

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



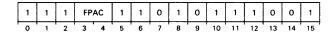
Stores a copy of the rightmost byte of ACD into memory at the address specified by ACS.

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.

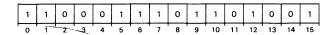
NOTES: If the number in the specified FPAC has any fractional part, the result of the instruction is undefined. Use the Integerize instruction (FINT) to clear any fractional part.

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.

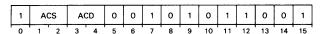
Upon successful termination, the contents of AC0 and AC1 remain unchanged; AC2 contains the original contents of AC3; and AC3 contains a byte pointer pointing to the first byte following the destination field. Carry is set to 0 and *overflow* is 0.

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

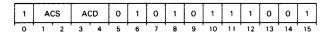


Subtracts two integers contained in accumulators.

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



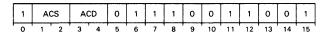
Tests a bit and skips if the bit is 0.

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.

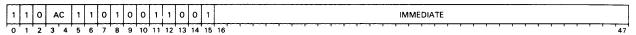
The instruction forms a bit pointer from the contents of ACS and ACD. ACS contains the high-order bits of the bit pointer; ACD contains the low-order bits. ACS and ACD can be specified to be the same accumulator; in this case, the specified accumulator supplies the low-order bits of the bit pointer. 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



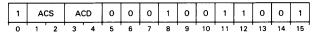
Compares one unsigned integer to another and skips if the first is less than or equal to the second.

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



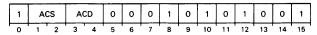
Compares one unsigned integer to another and skips if the first is greater than or equal to the second.

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



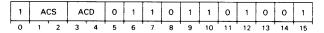
Compares one unsigned integer to another and skips if the first is greater than the second.

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

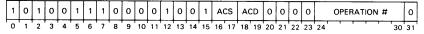


Exchanges the 32-bit contents of ACS and ACD.

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 200₈ 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 ACO, 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.

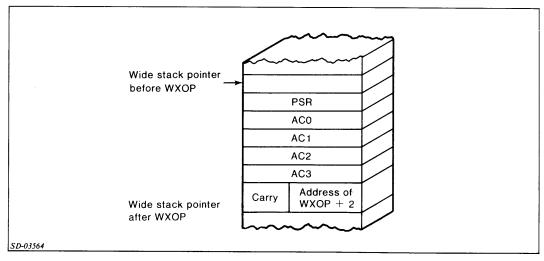


Figure 10.11 WXOP return block pushed

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)

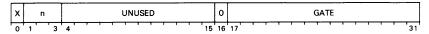
XCALL [@]displacement[,index[,argument count]]



Evaluates the address of a subroutine call.

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.

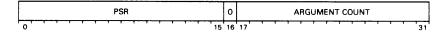
If the target address specifies an inward ring call, then the instruction assumes the target address has the following format:



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 contents of the PC in the return block are undefined.

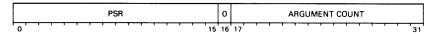
If the specified gate is legal, or if the target address specifies an intra-ring crossing, then the instruction loads the contents of the PC, plus three, into AC3. The contents of AC3 always reference the current segment.

If bit 0 of the argument count is 0, then the instruction creates a word with the following format:



The instruction pushes this word onto the wide stack. If a stack overflow occurs after this push, a stack fault occurs and no subroutine call is made. Note that the value of the PC in the return block is undefined.

If bit 0 of the argument count is 1, then the instruction assumes the top word of the wide stack has the following format:



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.

Exchange Accumulators

XCH acs.acd

*



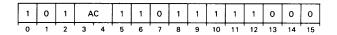
Exchanges the contents of two accumulators.

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.

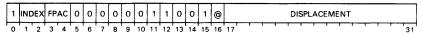
NOTES: If bits 16–31 of the specified accumulator contains the first word of a 2-word instruction, the word following the **XCT** instruction is used as the second word. Normal sequential operation then continues from the second word after the **XCT** instruction.

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]

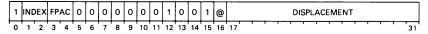


Adds the 64-bit floating-point number in the source location to the 64-bit floating-point number in FPAC and places the normalized result in FPAC.

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]



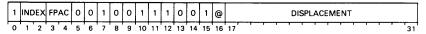
Adds the 32-bit floating-point number in the source location to the 32-bit floating-point number in FPAC and places the normalized result in FPAC.

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]

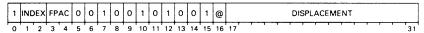


Divides the 64-bit floating-point number in FPAC by the 64-bit floating-point number in the source location and places the normalized result in FPAC.

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]



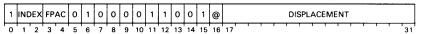
Divides the 32-bit floating-point number in bits 0–31 of FPAC by the 32-bit floating-point number in the source location and places the normalized result in FPAC.

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]



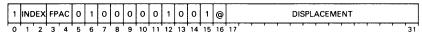
Moves four words out of memory and into a specified FPAC.

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]



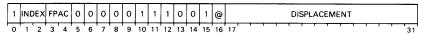
Moves two words out of memory into a specified FPAC.

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]

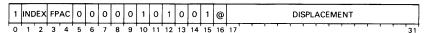


Multiplies the 64-bit floating-point number in the source location by the 64-bit floating-point number in FPAC and places the normalized result in FPAC.

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]

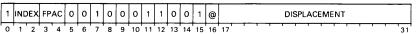


Multiplies the 32-bit floating-point number in the source location by the 32-bit floating-point number in bits 0-31 of FPAC and places the normalized result in FPAC.

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]

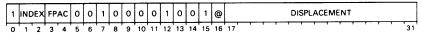


Subtracts the 64-bit floating-point number in the source location from the 64-bit floating-point number in FPAC and places the normalized result in FPAC.

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]

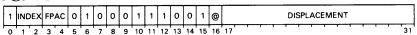


Subtracts the 32-bit floating-point number in the source location from the 32-bit floating-point number in bits 0-31 of FPAC and places the normalized result in FPAC.

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]



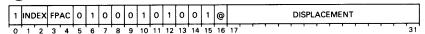
Stores the contents of a specified FPAC into a memory location.

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]



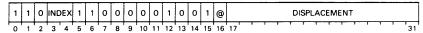
Stores the contents of a specified FPAC into a memory location.

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]



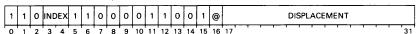
Loads an effective address into the program counter.

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]



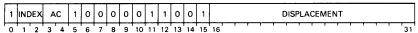
Saves a return address and transfers control to a subroutine.

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]

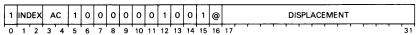


Calculates a byte pointer and loads the byte into the specified accumulator.

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]

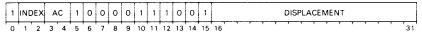


Loads an effective address into an accumulator.

Calculates the effective address, E. Loads E into the specified accumulator. Carry is unchanged and *overflow* is 0.

Load Effective Byte Address (Extended Displacement)

XLEFB ac, displacement [, index]



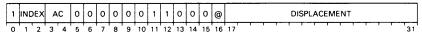
Loads an effective byte address into an accumulator.

Calculates the effective byte address. Loads the byte address into the specified accumulator. Carry is unchanged and *overflow* is 0.

NOTE: Index bits of 00 force the first address in the effective address calculation to be in the current segment of execution.

Narrow Add Memory Word to Accumulator (Extended Displacement)

XNADD ac,[@]displacement[,index]

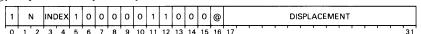


Adds an integer contained in a memory location to an integer contained n an accumulator.

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]

1	11	NDE	×	Α	С	0	0	0	1	1	0	1	1	0	0	0	@	DISPLACEMENT
6	_	1 ;	2 '	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17 31

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.

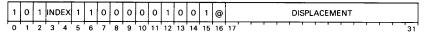
In either case, the instruction loads the incremented memory location into the AC.

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]



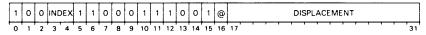
Decrements the contents of a location and skips the next word if the decremented value is zero.

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]



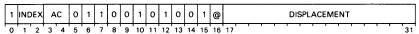
Increments the contents of a location and skips the next word if the incremented value is zero.

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]

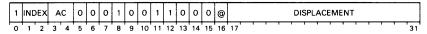


Sign extends and loads the contents of a memory location into an accumulator.

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]

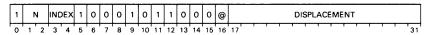


Multiplies an integer in memory by an integer in an accumulator.

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, $\lceil (a) \rceil$ displacement $\lceil index \rceil$



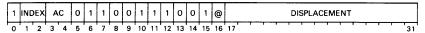
Subtracts an integer in the range of 1 to 4 from an integer contained in a 16-bit memory location.

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]

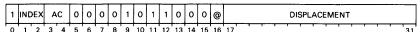


Stores the contents of an accumulator in a memory location.

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]

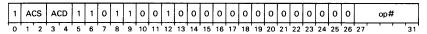


Subtracts an integer in memory from an integer in an accumulator.

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, ACO, 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.

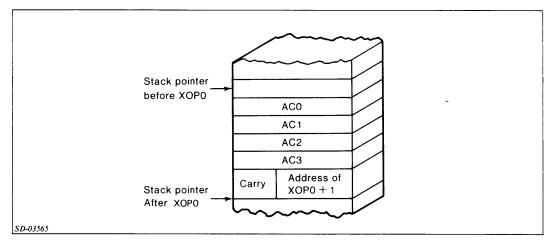
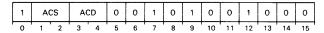


Figure 10.12 XOPO return block

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



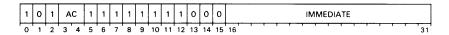
Exclusively ORs the contents of two accumulators.

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.

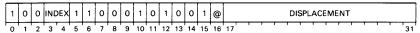
Forms the logical exclusive OR of the contents of the immediate field and the contents of bits 16–31 of the specified accumulator and places the result in bits 16–31 of the specified accumulator. Carry remains unchanged and *overflow* is 0.

Bits 0–15 of the modified accumulator are undefined after completion of this instruction.

*

Push Address (Extended Displacement)

XPEF [@]displacement[,index]

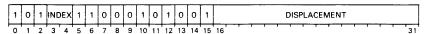


Pushes an address onto the wide stack.

Calculates the effective address, E. Pushes E onto the wide stack, then checks for stack overflow. Carry is unchanged and overflow is 0.

Push Byte Address (Extended Displacement)

XPEFB displacement[,index]

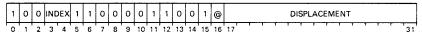


Pushes an effective byte address onto the wide stack.

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]



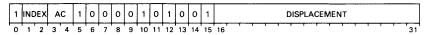
Pushes the program counter onto the wide stack and jumps to a subroutine.

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]

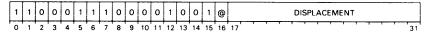


Stores the low-order byte of the specified accumulator in memory.

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 initializes AC0, AC1, AC2, PSR, and PC.

AC0 contains the revised priority mask.

AC1 contains the I/O channel and the device code in bits 23-31; zero extended.

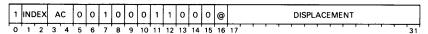
AC2 contains the entry zero address of the device control table.

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.

Wide Add Memory Word to Accumulator (Extended Displacement)

XWADD ac, (@) displacement [, index]

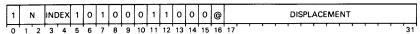


Adds an integer contained in memory to an integer contained in an accumulator.

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]



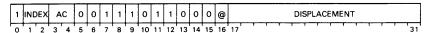
Adds an integer in the range of 1 to 4 to an integer contained in a 32-bit memory location.

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.

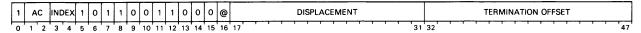
If the quotient is within the range of -2,147,483,648 to +2,147,483,647 inclusive, the instruction loads the quotient into the specified accumulator. Overflow is 0.

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.

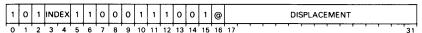
In either case, the instruction loads the incremented memory location into the AC.

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]



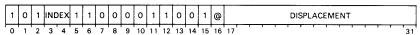
Decrements the contents of a location and skips the next word if the decremented value is zero.

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]



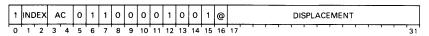
Increments the contents of a location and skips the next word if the incremented value is zero.

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]

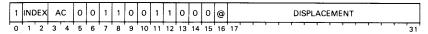


Loads the contents of a memory location into an accumulator.

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]



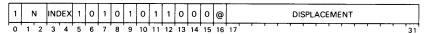
Multiplies an integer in an accumulator by an integer in memory.

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]



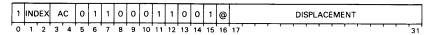
Subtracts an integer in the range of 1 to 4 from an integer contained in a 32-bit memory location.

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]

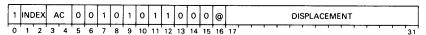


Stores the contents of an accumulator in a memory location.

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]



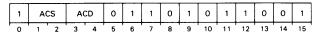
Subtracts an integer contained in memory from an integer contained in an accumulator.

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.

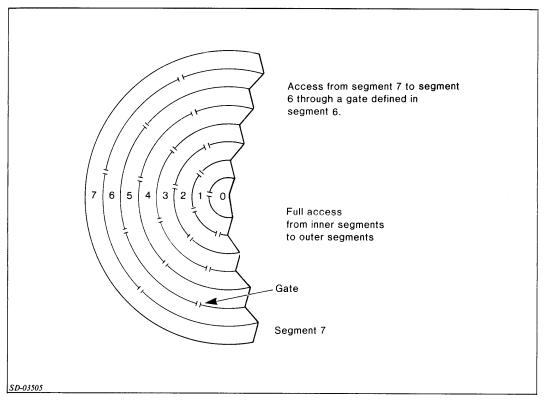


Figure 1.6 Virtual address space

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 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 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.

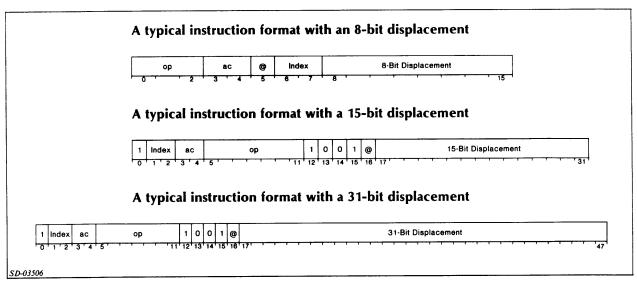


Figure 1.7 Memory reference instruction word addressing formats

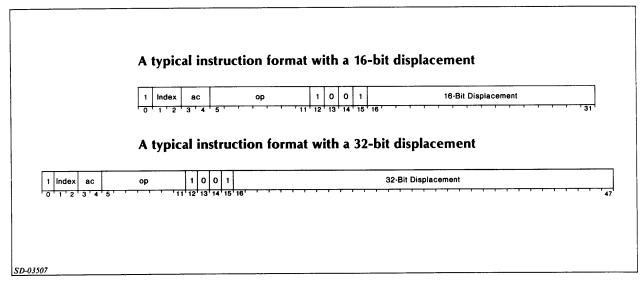


Figure 1.8 Memory reference instruction byte addressing formats

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.

Appendix A Instruction Summary

The Instruction Summary lists the 32-bit processor instructions alphabetically by assembler-recognizable mnemonic, giving the format, data type used, action performed, and location contents before and after instruction execution.

The C/350 compatible instructions are identified with an asterisk (*) located at the beginning of the instruction mnemonic.

The following abbreviations are used throughout this summary:

Abbreviation	Meaning
Data Indicators	
#	Integer
2#	2's complement integer
str	Byte string
bcd	Binary coded decimal
high	High order of number
low	Low order of number
Н	High number for comparison
L	Low number for comparison
Operators	
\rightarrow	Returned to
xxx	Complement of xxx
+	Addition
-	Subtraction or negation
/	Division
*	Multiplication
=	Equality
≠	Nonequality
< > ?	Less than
>	Greater than
	Comparison
=skip	If condition is true, skip the next sequential word
OR	Logical OR
AND	Logical AND
XOR	Logical Exclusive OR
n*4	Hex shift $(n*4 = bits shifted)$
?	Unpredictable result
&	Ties together two (or more) items to be operated upon as one

```
Fixed-Point
                             Fixed-point accumulator
    ac
                             Source ac
     acs
     acd
                             Destination ac
     PSR
                             Processor status register
                             CRY bit (PSR)
     CRY
                             OVR bit (PSR)
     OVR
     IRES
                             IRES bit (PSR)
     ALU CRY
                             Arithmetic and logic unit CRY bit
                             Arithmetic overflow
     overflow
Floating-Point
                             Floating-point number (fp#s=single precision, fp#d=double precision)
    fp#
     FPSR()
                             Floating-point status register (flags updated)
     fpac
                             Floating-point accumulator
                             Source fpac
     facs
     facd
                             Destination fpac
     abs
                             Absolute value
     int
                             Integer
                             Normalized number
     norm
                             New exponent
     new exp.
Decimal/Byte
    bp
                             Byte pointer
                             Ascending order
     asc.
     desc.
                             Descending order
                             Character (ASCII or decimal digit)
     char
     del.table addr
                             Delimiter table address
     trans.table bp
                             Translation table byte pointer
Stack
                             Wide stack pointer
    wsp
                             Wide frame pointer
     wfp
     wsl
                             Wide stack limit
                             Wide stack base
     wsb
                             Wide stack fault address
     wsa
                             Narrow stack pointer
     sp
                             Narrow frame pointer
     fp
     sl
                             Narrow stack limit
                             Narrow stack fault address
     sa
                             Double words
     d.words
Miscellaneous
    E
                             Calculated effective address
                             Source E
     S(E)
                             Destination E
     D(E)
                             Address in page zero
     (#) page zero
                             Unknown and soon to be lost
     unch
                             Unchanged
     displ.
                             Displacement
     PC
                             Program counter
     ION
                             Interrupt on flag
     R/W
                             Read/write command
      1's
                             Bits set to 1
     0's
                             Bits set to 0
     Q
                             Queue
     dec#
                             Decimal number
                             Modified page bit
     mod
     ref
                             Referenced page bit
     PTE
                             Page table entry
     SBR
                             Segment base register
```

NOTE: For all operations, unless specifically mentioned:

Before Instruction Execution: Upon Instruction Completion:

OVR=x unchanged
CRY=x unchanged
overflow=x unchanged
FPSR bits=x updated
BUSY, DONE flags=x unchanged

Instruction Format	Action	Before (Location =)	After (Location =)
*ADC[c][sh][#] acs,acd[.skip] NOTE: If result >65,535 then CRY—CRY	acs + acd →acd	acs == # acd == #	unchanged result
*ADD(c][sh][#] acs,acd[,skip] NOTE: If result >65,535 then CRY→CRY	acs+acd—acd	acs = # acd = #	unchanged result
*ADDI i,ac	ac+i—ac	ac = 2# i = 2#	result unchanged
*ADI n,ac	ac+n—ac	ac = # n = # (1-4)	result unchanged
*ANC acs,acd	acs AND acd—acd	acs = # acd = #	unchanged result
*AND[c][sh][#] acs,acd[,skip]	acs AND acd—acd	acs = # acd = #	unchanged result
*ANDI i,ac	ac AND i—ac	ac = # i = #	result unchanged
*BAM	S(E) + ACO—D(E)	AC0 = #(addend) AC1 = #(no. words) AC2 = source E AC3 = dest. E	unchanged O last E + 1 last E + 1
ВКРТ	wide return block →stack	stack = x	+6 double words
		PSR=x PC=(BKPT)	0 (10-11) page zero
*BLM	S(E)→D(E)	AC1=#(no. words) AC2=source E AC3=dest. E	0 last E + 1 last E + 1
*BTO acs.acd NOTE: If acs=acd, word pointer=0	1 →(E)bit	acs = word pointer acd = word offset & bit pointer (E)bit = x	unchanged unchanged 1
*BTZ acs,acd NOTE: If acs=acd, word pointer=0	O—(E)bit	acs=word pointer acd=word offset & bit pointer	unchanged unchanged
CIO acs,acd	R/W—I/O system bus	(E)bit=x acs=#(command) write	0 unchanged
		acd=# read acd=x	unchanged result
CIOI i,acs,acd	R/W→I/O system bus If acs=acd, i=command If acs≠acd, i OR acs=command	<pre>acs = #(command) i = #(command) write acd = #</pre>	unchanged unchanged unchanged
*CLM acs,acd	acs>=L & acs<=H = skip	read acd=x acs=2#	result unchanged
	If acs is acd	acd=(L) 2# (L+1)=H 2# acs=2#	unchanged unchanged unchanged
	405 15 404	(CLM + 1) = L (CLM + 2) = H	unchanged unchanged unchanged

Instruction Format	Action	Before (Location =)	After (Location =)
*CMP	string 1 ? string 2 Result Returned -1—AC1 (1<2)	AC0 = str2 #bytes (# = asc, 2# = desc) AC1 = str1 #bytes	O or # bytes left
	$0 \rightarrow AC1 (1=2)$ + $1 \rightarrow AC1 (1<2)$	(#=asc, 2#=desc) AC2=str2 bp AC3=str1 bp	last bp \pm 1 or failing byte last bp \pm 1 or failing byte
*CMT	S(E)→D(E) If byte = delimiter halt instruction and	AC0 = del.table addr. AC1 = # bytes (#=asc, 2#=desc)	E(del.table) 0 or unmoved bytes
46141	byte not moved	AC2 = destination bp AC3 = source bp	last bp + 1 last bp + 1
*CMV NOTE If S < D, Remainder of D is filled with spaces	S(E)→D(E) CRY=relative length 1→S>D	AC0 = dest. #bytes (# = asc, 2# = desc) AC1 = source #bytes	0 0 or unmoved bytes
	0→S= <d< td=""><td>(#=asc, 2#=desc) AC2=destination bp AC3=source bp CRY=x</td><td>last bp $+$ 1 last bp $+$ 1 relative length</td></d<>	(#=asc, 2#=desc) AC2=destination bp AC3=source bp CRY=x	last bp $+$ 1 last bp $+$ 1 relative length
*COB acs,acd NOTE: If acs is acd, acs=unchanged	acs(1's)+acd—acd	acs=# acd=2#	unchanged result
*COM[c][sh][#] acs,acd[,skip]	acs—acd	acs = # acd = x	unchanged result
CRYTC	<u>CRY</u> —CRY	CRY=CRY	CRY
CRYTO	1—CRY	CRY=CRY	1
CRYTZ	0—CRY	CRY=CRY	0
*CTR	S(E)—D(E) translates	AC0=trans.table bp AC1=# bytes (2#) AC2=destination bp AC3=source bp	unchanged O last bp $+$ 1 last bp $+$ 1
	OR	·	·
	string1 ? string2 translates -1AC1 (1<2) 0AC1 (1=2) +1AC1 (1>2)	AC0 = trans.table bp AC1 = # bytes (#) AC2 = string2 bp AC3 = string1 bp	unchanged result code last bp \pm 1 or failing byte last bp \pm 1 or failing byte
CVWN ac NOTE: If ac(bits 0-16) \neq all 1's or all 0's before conversion, Overflow = 1	ac(32 bits)—ac(16 bits)	ac = #(32-bit)	#(16-bit)
*DAD acs,acd	acs+acd+CRY—acd	acs=bcd acd=bcd CRY=x	unchanged result Decimal carry
DEQUE	Queue-element—Queue	ACO = E(Q descriptor) AC1 = E(data element) Q descriptor	unchanged unchanged updated
*DHXL n,ac	ac∾+1(shift left(n*4)) →ac∾+1	ac = high ac + 1 = low n = # (0-8)	result result unchanged
*DHXR n,ac	ac∾+1(shift right(n*4)) →ac∾+1	ac = high ac + 1 = low n = # (0-8)	result result unchanged
*DIA[f] ac,device	device(A buffer)—ac	A buffer = # ac = x BUSY,DONE flags = x	unchanged result [f]
*DIB[f] ac,device	device(B buffer)—ac	B buffer = # ac = x BUSY,DONE flags = x	unchanged result [f]
*DIC[f] ac,device	device(C buffer)—ac	C buffer = # ac = x BUSY,DONE flags = x	unchanged result [f]

Instruction Format	Action	Before (Location =)	After (Location =)
*DIV	AC0&AC1/AC2→	ACO=#(high dividend)	#(remainder)
NOTE: If $ACO > = AC2$ then $1 \rightarrow CRY$,	AC1(quotient)	AC1=#(low dividend)	#(quotient)
ACO&ACI = unchanged, instruction terminates	•	AC2 = #(divisor)	unchanged
Acocaci unchangea, instruction terminates	Additional	CRY=x	0
+0.110			•
*DIVS	ACO&AC1/AC2→	ACO=2#(high dividend)	2#(remainder)
NOTE: If quotient overflows, $l \rightarrow CRY$,	AC1(quotient)	AC1 = 2#(low dividend)	2#(quotient)
AC0&AC1=?, instruction terminates	ACO(remainder)	AC2 = 2#(divisor)	unchanged
		CRY=x	0
*DIVX	AC0&AC1/AC2→	AC0=sign of AC1	2#(remainder)
NOTE: If quotient overflows, $l \rightarrow CRY$,	AC1(quotient)	AC1 = 2#(dividend)	2#(quotient)
AC0&AC1=?, instruction terminates	ACO(remainder)	AC2 = 2#(divisor)	unchanged
,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,		CRY=x	0
*DICH			_
*DLSH acs,acd	acd∿ + 1 (shift left/right)	acs = 2# for shift	unchanged
	—acd∿ + 1	(+= left/-= right)	
		acd=high	result
		acd+1=low	result
*DOA[f] ac,device	ac—device(A buffer)	ac = #	unchanged
-107		A buffer $= x$	result
		BUSY,DONE flags=x	[f]
*DOD[f] as davisa	no dovino/P buffor\	ac = #	
*DOB[f] ac,device	ac—device(B buffer)		unchanged
		B buffer = x	result
		BUSY,DONE flags = x	[f]
*DOC[f] ac,device	ac—device(C buffer)	ac = #	unchanged
,		C buffer = x	result
		BUSY, DONE flags = x	[f]
*DSB acs,acd	acd-acs- CRY —acd	acs=bcd	unchanged
· · · · · · · · · · · · · · · · · · ·	acu-acs-ch i →acu	acd=bcd	•
NOTE: If CRY=0, result is -;			result
if CRY=1, result is +		CRY=x	Decimal borrow
*DSPA ac,[@]displacement[,index]	if ac <l ac="" or="">H</l>	E = (dispatch table)	unchanged
	then (DSPA) + 1—PC	E-2=L(2#)	unchanged
		E-1 = H(2#)	unchanged
		ac=2#	unchanged
	If ac $>=$ L and ac $<=$ H	Last table word=	
	then E-L+#—PC	E + 2*(H-L)	unchanged
	If $(E-L+\#)=177777_8$	CRY=x	0
	(DSPA) ± 1→PC		
*DC7 (@ldianlasament) index)	(E) 1 (E)	(E) = #	#-1
*DSZ [@]displacement[,index]	(E)-1→(E)	(E) — #	## = I
	If $(E) = 0 = skip$		
DSZTS	(wsp)-1—(wsp)	(wsp) = #	#-1
	If $(wsp) = 0 = skip$		
ECLID	CPU id→AC0	ACO=x	CPU id
*EDIT		ACO=bp	Р
		(1st subopcode)	
		AC1 = (data-type)	?
		AC2=bp (dest.)	DI
		AC3=bp (source)	SI
		OVR = x	0
		CRY=x	Т
		T=0	?
		S=x	Sign $(0 = +, 1 = -)$
NOTE: For subopcodes		SI=AC3	last bp + 1
If $j(high-order\ bit)=1$, word is at $(sp+1+j)$		DI=AC2	last bp + 1
If $j(high-order\ bit)=0$, word is at j		P=ACO	last bp + 1
DADI p0	DI+p0→DI	p0=2#	•
•		·	unchanged
DAPS $p\theta$	If $S=0$, $P+p0\rightarrow P$	p0=2#	unchanged
		P = (DAPS)	?
		I TOAI SI	f

Instruction Format	Action	Before (Location =)	After (Location =)
DAPT p0	If T = 1, P+p0—P	p0=2# P=(DAPT) T=x	unchanged ? unchanged
DAPU p0	$P+p0\longrightarrow P$	p0=2# P=(DAPU)	unchanged (DAPU) + 2#
DASI p0	SI+p0—SI	p0=2#	unchanged
DDTK $k,p0$ NOTE: If $(k) = -$, stack word = $sp + l + k$; If $(k) = +$, stack word = $fp + l + k$	$(k)-1 \longrightarrow (k)$ If $(k) \neq 0$, $P+p0 \longrightarrow P$	(k) = # p0 = 2# P = (DDTK)	#-1 unchanged ?
DEND	Stop Edit(ing)		
DICI $n,p0[,p1,,p(n-1)]$	pO—(DI) p1—(DI+1) 	n=# or 2# p0[p1,]=character P=P	unchanged unchanged P + n + 2
	$p(n-1) \rightarrow (DI + n)$	DI = DI	DI+n
DIMC j,p0	p0—(DI) p0—(DI+j)	j=# or 2# p0=character DI=DI	unchanged unchanged DI + j
DINC p0	p0→(DI)	p0 = character DI = DI	unchanged DI + 1
DINS p0,p1	If $S=0$, $p0\longrightarrow(DI)$ If $S=1$, $p1\longrightarrow(DI)$	p0 = character p1 = character DI = DI S = x	unchanged unchanged DI + 1 unchanged
DINT p0.p1	If $T = 0$, $pO \rightarrow (DI)$ If $T = 1$, $p1 \rightarrow (DI)$	p0 = character p1 = character D1 = D1 T = x	unchanged unchanged DI + 1 unchanged
DMVA j	(SI)—(DI) (SI+1)—(DI+1) (SI+j)—(DI+j)	j=# or 2# SI=SI DI=DI T=x	unchanged SI+j DI+j 1
DMVC j	If source data type=3 and j>0 (SI)→(DI) (SI+1)→(DI+1)	j=# or 2# SI=SI DI=DI T=x	unchanged SI+j DI+j 1
	 (SI+j)→(DI+j)		
DMVF j,p0,p1,p2	If source data type=3 and j>0 and (SI)=source # sign SI+1→SI If T=1 (SI)→(DI) (SI+1)→(DI+1)	j=# or 2# p0=character p1=character p2=character T=x S=x	unchanged unchanged unchanged unchanged ? unchanged
	(Si ↑ 1)—(Di ↑ 1) 	SI=SI	SI+j
	$(SI + j) \longrightarrow (DI + j)$ If T = 0 If (SI) = 0(or space) $p0 \longrightarrow (DI)$ If (SI) \neq 0 $1 \longrightarrow T$ If S = 0 $p1 \longrightarrow (DI)$ If S = 1	DI = DI DI = DI	DI+j DI+j+1
DMVN j	p2—(DI) If source data type=3 and j>0 (SI)—(DI) (SI+1)—(DI+1)	j=# or 2# SI=SI DI=DI T=x	unchanged SI+j DI+j 1
	 (SI + j)→(DI + j)		

Instruction Format	Action	Before (Location =)	After (Location =)
DMVO p0,p1,p2,p3	If source data type=3 and (SI)=source # sign SI+1 \rightarrow SI If (SI)=0(or space) If S=0, p0 \rightarrow (DI) If S=1, p1 \rightarrow (DI)	p0,p1,p2,p3 = character SI = SI DI = DI S = x T = x	unchanged SI + 1 DI + 1 unchanged unchanged
	If $(SI) \neq 0$ If $S = 0$, $p2 + (SI) \rightarrow (DI)$ If $S = 1$, $p3 + (SI) \rightarrow (DI)$	T=x	1
DMVS j.p0	If source data type=3 and $j>0$ and (Si) = source # sign $Si+1 \rightarrow Si$ $(Si) \rightarrow (Di)$ $(Si+1) \rightarrow (Di+1)$ 	j=# or 2# p0=character SI=SI DI=DI T=x	unchanged unchanged SI+j DI+j ?
	 (SI+j)→(DI+j) If T=0 p0→0s & spaces		
DNDF <i>p0,p1</i>	If $T = 1$, $DI \rightarrow DI$	p0 = character p1 = character DI = DI	unchanged unchanged unchanged
	If $T=0$ If $S=0$, $p0\rightarrow(DI)$ If $S=1$, $p1\rightarrow(DI)$	T = x $S = x$ $Di = Di$	1 unchanged DI + 1
DSSO	1—S	S=x	1
DSSZ	0 ⊸ S	S=x	0
DSTK $k,p0$ NOTE: $If(k) = +, (k) = sp + l + k;$ If(k) = -, (k) = fp + l + k	p 0 —(k)	p0 = character $(k) = 2\#$	unchanged unchanged
DSTO	1—→T	T=x	1
DSTZ	0 ⊤	T=x	0
*EDSZ [@]displacement[,index]	$(E)-1 \longrightarrow (E)$ If $(E) = 0 = \text{skip}$	(E) = #	#-1
*EISZ [@]displacement[,index]	$(E) + 1 \rightarrow (E)$ If $(E) = 0 = \text{skip}$	(E) = #	#+1
*EJMP [@]displacement[,index]	E⊸PC	PC = (EJMP)	E
*EJSR [@]displacement[,index]	E⊸PC	PC=(EJSR) AC3=x	E PC+2
*ELDA ac,[@]displacement[,index]	(E)—ac	ac = x (E) = #	# unchanged
*ELDB ac,displacement[,index]	(E)→ac(right)	ac = x (E) = # CRY = x	# unchanged O
*ELEF ac,[@]displacement[,index]	E—ac	ac=x	E
ENQH	data element→Q Head	AC0=E(Q descriptor) AC1=E(Q element) AC2=E(element to add)	unchanged unchanged unchanged
ENQT	data element⊶Q Tail	AC0=E(Q descriptor) AC1=E(Q element) AC2=E(element to add)	unchanged unchanged unchanged
*ESTA ac,[@]displacement[,index]	ac—(E)	(E) = x $ac = #$	ac unchanged
*ESTB ac,displacement[,index]	ac(right)(E)	(E) = x $ac = #$	ac(right) unchanged
FAB fpac	absolute(fpac)fpac	fpac=fp# FPSR(N,Z)	abs(fp#) N=0 Z=updated

Instruction Format	Action	Before (Location =)	After (Location =)
*FAD facs.facd	facs + facd → facd	facs=fp#d facd=fp#d	unchanged fp#d
*FAMD fpac./@]displacement[,index]	(E) +fpac→fpac	(E) = fp#d fpac = fp#d	unchanged fp#d
*FAMS fpac,[@]displacement[,index]	(E) + fpac→fpac	(E) = fp#s fpac = fp#s	unchanged fp#s
*FAS facs.facd	facs + facd → facd	facs=fp#s facd=fp#s	unchanged fp#s
*FCLE	0FPSR(0-4)	FPSR(0-4) = x	0
*FCMP facs facd	facs ? facd	facs = fp# facd = fp# FPSR(N,Z) = x	unchanged unchanged (N Z) 0 1 facs=facd 1 0 facs>facd 0 0 facs <facd< td=""></facd<>
*FDD facs facd	facd/facs—facd	facs=fp#d facd=fp#d	unchanged fp#d
*FDMD fpac,[@]displacement[,index]	fpac/(E)—fpac	(E) = $fp#d$ fpac = fp#d	unchanged fp#d
*FDMS fpac,[@]displacement[,index]	fpac/(E)—fpac	(E) = $fp#s$ fpac = fp#s	unchanged fp#s
*FDS facs,facd	facd/facs—facd	facs=fp#s facd=fp#s	unchanged fp#s
*FEXP fpac NOTE: If fpac=true 0, fpac=unchanged	ACO—exp(fpac)	AC0 = # fpac = fp# FPSR(N,Z) = x	unchanged fp#(new exponent) updated
*FFAS ac.fpac	integer(fpac)—ac	fpac=fp# ac=x FPSR(N,Z)=x (MOF)=x	unchanged 2# unch updated
*FFMD fpac,[@]displacement[,index]	integer(fpac)—(E)	fpac = fp# $(E) = x$ $FPSR(N,Z) = x$ $(MOF) = x$	unchanged int(2#)(32-bit) unchanged updated
*FHLV fpac	fpac/2—fpac	fpac = fp# $FPSR(N,Z,UNF) = x$	result updated
*FINT fpac NOTE: FINT truncates towards 0 with no rounding	integer(fpac)—fpac	fpac=fp#	result
*FLAS ac fpac	ac→fpac	ac = 2# $fpac = x$ $FPSR(N,Z) = x$	unchanged fp#s updated
*FLDD fpac,[@]displacement[,index]	(E)—fpac	(E) = fp#d $fpac = x$	unchanged fp#d
*FLDS fpac,[@]displacement[,index]	(E)—fpac	(E) = fp#s fpac = x	unchanged fp#s
*FLMD fpac,[@]displacement[,index]	(E)—fpac	(E) = 2# (32-bit) fpac = x FPSR(N,Z) = x	unchanged fp#d updated
*FLST [@]displacement[,index]	(E)→FPSR	(E) = fp#s FPSR(all) = x	unchanged (E)
*FMD facs.facd	facd*facs—facd	facs=fp#d facd=fp#d	unchanged fp#d
FMMD fpac,[@]displacement[,index]	fpac(E)—fpac	(E)=fp#d fpac=fp#d	unchanged fp#d
FMMS [fpac,[@]displacement[,index]	fpac(E)—fpac	(E)=fp#s fpac=fp#s	unchanged fp#s
*FMOV facs.facd	facs—facd	facs=fp# facd=x	unchanged facs

*

Instruction Format	Action	Before (Location =)	After (Location =)
*FMS facs.facd	facd*facs—facd	facs=fp#s facd=fp#s	unchanged fp#s
*FNEG fpac NOTE: True 0 is unchanged.	-fpac—fpac	fpac=fp#	-fp#
*FNOM fpac	norm(fpac)—fpac	fpac=fp# FPSR(N,Z,UNF)=x	norm fp# updated
*FNS	PC+1→PC	PC=PC	PC+1
*FPOP	-18 words—stack	stack = stack FPAC3 = x FPAC2 = x FPAC1 = x FPAC0 = x FPSR = x	-18 words 1st 4 2nd 4 3rd 4 4th 4 last 2
*FPSH	+18 words→stack	stack = stack 1st 2 = x 2nd 4 = x 3rd 4 = x 4th 4 = x 5th 4 = x	+ 18 words FPSR FPACO FPAC1 FPAC2 FPAC3
FRDS facs.facd NOTE: FPSR(8) is ignored for rounding	facs(fp#d)—facd(fp#s)	facs=fp#d facd=x	fp#d fp#s
*FRH fpac	fpac(high 16)—ACO	fpac = fp# $ACO = x$ $FPSR(N,Z) = x$	unchanged high 16(fp#) unchanged
*FSA	PC+2→PC	PC=PC	PC+2
*FSCAL fpac	ACO-fpac(exp) = fpac(mantissa shift) ACOfpac(exp)	ACO = # $fpac = fp#$ $FPSR(N,Z) = x$	unchanged result updated
*FSD facs.facd	facd-facs—facd	facs=fp#d facd=fp#d	unchanged fp#d
*FSEQ	If $FPSR(Z) = 1 = skip$	FPSR(Z) = x	unchanged
*FSGE	If $FPSR(N) = 0 = skip$	FPSR(N) = x	unchanged
*FSGT	If $FPSR(Z&N) = 0 = skip$	FPSR(N,Z) = x	unchanged
*FSLE	If $FPSR(ZorN) = 1 = skip$	FPSR(N,Z) = x	unchanged
*FSLT	If $FPSR(N) = 1 = skip$	FPSR(N) = x	unchanged
*FSMD fpac,[@]displacement[,index]	fpac-(E)—fpac	(E)=fp#d fpac=fp#d	unchanged fp#d
*FSMS fpac,[@]displacement[,index]	fpac-(E)—fpac	(E)=fp#s fpac=fp#s	unchanged fp#s
*FSND	If $FPSR(DVZ) = 0 = skip$	FPSR(DVZ) = x	unchanged
*FSNE	If $FPSR(Z) = 0 = skip$	FPSR(Z) = x	unchanged
*FSNER	If $FPSR(1-4) = 0 = skip$	FPSR(1-4)=x	unchanged
*FSNM	If $FPSR(MOF) = 0 = skip$	FPSR(MOF) = x	unchanged
*FSNO	If $FPSR(OVF) = 0 = skip$	FPSR(OVF) = x	unchanged
*FSNOD	If $FPSR(OVF\&DVZ) = 0 = skip$	FPSR(OVF,DVZ) = x	unchanged
*FSNU	If $FPSR(UNF) = 0 = skip$	FPSR(UNF) = x	unchanged
*FSNUD	If $FPSR(UNF\&DVZ) = 0 = skip$	FPSR(UNF,DVZ) = x	unchanged
*FSNUO	If FPSR(UNF&OVF)=0 =skip	FPSR(UNF,OVF) = x	unchanged
*FSS facs.facd	facd-facs—facd	facs=fp#s facd=fp#s	unchanged fp#s
*FSST [@]displacement[,index]	FPSR→(E)	(E) = x FPSR(all) = x	FPSR unchanged
*FSTD fpac, [@] displacement[, index]	fpac→(E)	fpac=fp#d (E)=x	unchanged fp#d
*FSTS fpac,[@]displacement[,index]	fpac(E)	fpac=fp#s (E)=x	unchanged fp#s

Instruction Format	Action	Before (Location =)	After (Location =)
*FTD	O→FPSR(TE)	FPSR(TE) = x	0
*FTE	1→FPSR(TE)	FPSR(TE)=x	1
FXTD	0—OVK	OVK = x	0
FXTE	1—OVK	OVK = x	1
TATE	0—OVR	OVR = x	0
*HALT NOTE: HALT=DOC 0,CPU	Stops the processor	ION flag=x	unchanged
*HLV ac NOTE: HLV rounds toward 0	ac/2—ac	ac = 2#	result
*HXL n,ac	ac(left shift(n*4))—ac	n = #(1-4) ac = #	unchanged result
*HXR n,ac	ac(right shift(n*4)—ac	n = #(1-4) ac = #	unchanged result
*INC[c][sh][#] acs,acd[,skip] NOTE: If acs>1777778, CRY—CRY	acs+1→acd	acs = # acd = x	unchanged result
*INTA ac NOTE: INTA ac = DIB ac,CPU	device code—ac	ac = x ION flag = x	device code unchanged
*INTDS NOTE: INTDS=NIOC CPU	O—ION flag	ION flag=x	0
*INTEN NOTE: INTEN=NIOS CPU	1—ION flag	ION flag=x	1
*IOR acs,acd	acs OR acd—acd	acs = # acd = #	unchanged result
*IORI i,ac	i OR ac→ac	i = # ac = #	unchanged result
*IORST · NOTE: IORST=DICC 0,CPU	Clear all I/O devices 0—priority mask	ION flag=x BUSY,DONE flags=x	0 0
*ISZ [@]displacement[,index]	$(E) + 1 \longrightarrow (E)$ If $(E) = 0 = \text{skip}$	(E) = #	#+1
ISZTS	$(wsp) + 1 \rightarrow (wsp)$ If $(wsp) = 0 = skip$	(wsp) = 32#	+ 1
*JMP [@]displacement[,index]	E⊸PC	PC=PC	E
*JSR [@]displacement[,index]	E→PC	PC=PC AC3=x	E PC + 1
LCALL [@]disp[,index][arg count]]	If E≡valid, E—PC	PC = PC AC3 = x OVR = x	E PC+4 O
LCPID	CPU id→AC0	ACO=x	CPU id
*LDA ac,[@]displacement[,index]	(E)—ac	ac=x	(E)
LDAFP	wfp—ac	ac = x wfp = #	wfp unchanged
LDASB	wsb→ac	ac=x wsb=#	wsb unchanged
LDASL	wsl—ac	ac = x wsl = #	wsl unchanged
LDASP	wsp→ac	ac=x wsp=#	wsp unchanged
LDATS	(wsp)—ac	ac = x (wsp) = #	(wsp) unchanged
*LDB acs,acd	(E)—acd	acs = bp acd = x (E) = byte	unchanged byte unchanged
*LDI fpac NOTE: A -0 sets fpac to true 0	dec#(E)—fp#(fpac)	(E) = decimal # fpac = x AC1 = data indicator AC2 = x AC3 = bp FPSR(N,Z)	unchanged norm fp# unchanged AC3 ? updated

Instruction Format	Action	Before (Location =)	After (Location =)

*LDIX	$dec\#(E) \longrightarrow fp\#(FPAC0, 1, 2, 3)$	FPAC0=x	1st unit
		FPAC1=x	2nd unit
		FPAC2 = x	3rd unit
		FPAC3 = x	4th unit
		FPSR(N,Z)	?
		AC1 = data indicator	unchanged
		AC2 = x	AC3
		AC3=bp	?
LDSP ac,[@]displacement[,index]	If ac <l ac="" or="">H</l>	E = dispatch table	unchanged
	then (LDSP) +2→PC	E-4=L(2#)	unchanged
		E-2 = H(2#)	unchanged
		AC=2#	unchanged
	If ac $>=$ L and ac $<=$ H		anonange a
	then E-2*(L-#)—PC		
	If E-2*(L-#)=		
	377777777 ₈ ,		
	then (LDSP) + 2—PC		
*LEF ac,/@/displacement/,index/	,		-
1011	E→ac	ac=x	E
LFAMD fpac,[@]displacement[,index]	(E) $+$ fpac \rightarrow fpac	(E) == fp#d	unchanged
		fpac=fp#d	fp#d
LFAMS fpac,[@]displacement[,index]	(E) + fpac→fpac	(E) = fp#s	unchanged
7. 10, 1		fpac=fp#s	fp#s
LFDMD fpac,[@]displacement[,index]	fpac/(E)—fpac	(E) = fp#d	•
LFDIVID Jpac, [w] uispiacement[,inaex]	ipac/(E/→ipac	•	unchanged
		fpac=fp#d	fp#d
LFDMS fpac,[@]displacement[,index]	fpac/(E)—fpac	(E) = fp#s	unchanged
		fpac=fp#s	fp#s
LFLDD fpac,[@]displacement[,index]	(E)—fpac	(E) = fp#d	unchanged
J1 11 C) 1	, , , , , ,	fpac=fp#d	fp#d
IFIDS frac [@]displacement[index]	/E) fnoo	•	·
LFLDS fpac,[@]displacement[,index]	(E)—fpac	(E) = fp #s	unchanged
		fpac=fp#s	fp#s
LFLST [@]displacement[,index]	(E)—FPSR	(E) = fp#d	unchanged
		FPSR(all) = x	(E)
LFMMD fpac,[@]displacement[,index]	fpac*(E)—fpac	(E) = fp#d	unchanged
		fpac=fp#d	fp#d
LFMMS fpac,[@]displacement[,index]	fpac*(E)fpac	(E) = fp#s	unchanged
Li Mino jpac, [wjaispiacement], macx	ipae (E)—ipae	fpac=fp#s	fp#s
A DRAGIN C. (C) III I			·
LFMSD fpac,[@]displacement[,index]	fpac-(E)—fpac	(E) = fp#d	unchanged
		fpac=fp#d	fp#d
LFSMS fpac,[@]displacement[,index]	fpac-(E)—fpac	(E) = fp#s	unchanged
·		fpac=fp#s	fp#s
LFSST [@]displacement[,index]	FPSR—(E)	(E) = x	FPSR
22 00 1 (@junsprucement,inuex)	,, 5,, ,(2)	FPSR(all) = x	unchanged
LECTO Garage (Coldinal and Coldinal	(5)		· ·
LFSTD fpac,[@]displacement[,index]	fpac—(E)	fpac=fp#d	unchanged
		(E) = x	fp#d
LFSTS fpac,[@]displacement[,index]	fpac—(E)	fpac=fp#s	unchanged
		(E) = x	fp#s
LJMP [@]displacement[,index]	E⊸PC	PC=PC	E
LJSR [@]displacement[,index]	E→PC	PC=PC	E
[w]uispiacement,timex]		AC3=x	PC+3
IIDD - Poul - Color	(5)		
LLDB ac,displacement[,index]	(E)—ac	ac=x	byte
		(E) = byte	unchanged
LLEF ac,[@]displacement[,index]	E⊸ac	ac=x	E
LLEFB ac, displacement [, index]	E⊸ac	ac=x	E(byte)
,			
LMRF	page(mod&ref bits)—AC0	AC0=x	mod&ref bits
		AC1 = pageframe #	unchanged
		modified bit=x	unchanged
		referenced bit = x	0

Instruction Format	Action	Before (Location =)	After (Location =)
LNADD ac.[@]displacement[.index] NOTE: If ALU overflows, Overflow=	(E) + ac →ac	(E) = # ac = # CRY = x	unchanged result ALU CRY
LNADI n,[@]displacement[.index] NOTE: If ALU overflows, Overflow=1	n + (E) → (E)	(E) = # n = # (1-4) CRY = x	result unchanged ALU CRY
LNDIV $ac./@/displacement/.index/$ NOTE: If $(E)=0$, or result >32.767 or <-32.768 , Overflow $=1$	ac/(E)—ac	(E) = # ac = #	unchanged result
LNDO ac.term offset,[@]disp[.index] NOTE: If ALU overflows, Overflow=1	(E) + 1 — (E) If (E) > ac then PC + 1 + termination offset — PC (E) — ac	(E) = # ac = # CRY = x	unchanged (E) ALU CRY
LNDSZ [@]displacement[,index]	$(E)-1 \longrightarrow (E) = 0$ If $(E) = 0 = \text{skip}$	(E) = #	(E)-1
LNISZ [@]displacement[,index]	$(E) + 1 \rightarrow (E)$ If $(E) = 0 = \text{skip}$	(E) = #	(E) + 1
LNLDA ac, [@] displacement [, index]	(E)—ac	(E) = # ac = x	unchanged (E)
LNMUL ac,[@]displacement[,index] NOTE: If result > 32,767 or <-32,768, Over- flow=1	(E)*ac→ac	(E) = # ac = #	unchanged result
LNSBI n,[@]displacement[,index] NOTE: If ALU overflows, Overflow=1	(E)-n—(E)	(E) = # n = # (1-4) CRY = x	result unchanged ALU CRY
LNSTA ac,[@]displacement[,index]	ac(E)	ac = # (E) = x	unchanged #
LNSUB ac,[@]displacement[,index] NOTE: If ALU overflows, Overflow=1	ac-(E)—ac	ac = # (E) = #	result unchanged
*LOB acs,acd	acs(# of Os) + acd—acd	acs = # acd = 2#	unchanged result
LPEF [@]displacement[,index]	Ewide stack	stack = stack	+E
LPEFB displacement[,index]	Ewide stack	stack = stack	+E(byte)
LPHY NOTE: If PTE ≠ page or validity fault, next word skipped	logical—physical	AC0=x AC1=logical address AC2=x	PTE unchanged physical address
LPSHJ [@]displacement[,index]	PC+3→wide stack E→PC	PC=PC stack=stack	E +(PC+3)
LPSR	PSR—ACO	PSR = x ACO = x OVR = x CRY = x	unchanged PSR unchanged unchanged
*LRB acs,acd NOTE: If ACS=ACD, then no count taken but 0—leading 1	acs(# of Os)+acd—acd O—high 1(acs)	acs=# acd=2#	new # result
LSBRA	new values—SBR(all)	SBR(0-7) = x $AC0 = E$	new unchanged
LSBRS	new values—SBR(1-7)	SBR(1-7) = x $SBR(0) = x$ $AC0 = E$	new unchanged unchanged
*LSH acs,acd	acd(shifted)—acd (acs determines shift)	acs = 2# (+=left,-=right)	unchanged
		acd=#	result

Instruction Format	Action	Before (Location =)	After (Location =)
*LSN	(E)=(non-0 or 0, + or -) +1 \rightarrow AC1=(+non-0) -1 \rightarrow AC1=(-non-0) 0 \rightarrow AC1=(+0) -2 \rightarrow AC1=(-0)	AC1 = data type AC2 = x AC3 = bp (E) = decimal #	value code AC3 ? unchanged
LSTB ac,displacement[,index]	ac—(E)	(E) = x ac = byte	byte unchanged
LWADD ac,[@]displacement[.index] NOTE: If ALU overflows, Overflow=1	ac+(E)—ac	(E) = # ac = # CRY = x	unchanged result ALU CRY
LWADI n./@/displacement[.index] NOTE: If ALU overflows, Overflow=1	(E) + n→(E)	(E) = # n = # (1-4) CRY = x	result unchanged ALU CRY
LWDIV $ac, \lfloor @/displacement \rfloor, index \rfloor$ NOTE: If quotient <-2,147,483,648 or >+2,147,483,647, or (E)=0; Overflow=1, and ac=unchanged	ac/(E)—ac(quotient)	(E) = # ac = # CRY = x	unchanged quotient ALU CRY
LWDO ac,term offset,[@]disp[,index] NOTE: If ALU overflows, Overflow=1	(E) + 1→(E) If (E) > ac then PC + 1 + termination offset →PC If (E) < = ac then PC + 1 →PC (E) →ac	(E) = # ac = # CRY = x	#+1 (E)+1 ALU CRY
LWDSZ [@]displacement[,index]	(E)-1=(E) If (E)=0 = skip	(E) = #	#-1
LWISZ [@]displacement[,index]	(E) + 1 = (E) If $(E) = 0 = skip$	(E) = #	#+1
LWLDA ac,[@]displacement[,index]	(E)—ac	(E) = # ac = x	unchanged (E)
LWMUL ac,[@]displacement[.index] NOTE: If result >-2,147,483,648 or <+2,147,483,647, Overflow=1	(E)*acac	(E) = # ac = #	unchanged result
LWSBI n,[@]displacement[,index] NOTE: If ALU overflows, Overflow=1	(E)-n—(E)	(E) = # n = # (1-4) CRY=x	#-n unchanged ALU CRY
LWSTA ac,[@]displacement[,index]	ac→(E)	ac = # (E) = x	unchanged ac
LWSUB ac./@/displacement[.index] NOTE: If ALU overflows, Overflow=1	ac-(E)—ac	ac = # (E) = # CRY = x	result unchanged ALU CRY
*MOV/c]/sh]/#] acs,acd/,skip/	acs—acd	acs = # acd = # CRY = x	unchanged acs ALU CRY
*MSKO ac NOTE: MSKO ac=DOB ac,CPU	ac—priority mask	ac = # ION flag = x mask = x	unchanged unchanged ac
*MSP ac	sp+ac—sp	ac = 2# sp = #	unchanged result
*MUL	AC1*AC2+AC0—AC0&AC1	ACO=# AC1=# AC2=#	result(high) result(low) unchanged
*MULS	AC1*AC2+AC0—AC0&AC1	AC1=2# AC1=2# AC2=2#	result(high) (sign—bit 16) result(low) unchanged
NADD acs,acd NOTE: If ALU overflows, Overflow=1	acs+acd—acd	acs=# acd=# CRY=x	unchanged result ALU CRY

Instruction Format	Action	Before (Location =)	After (Location =)
NADDI i,ac NOTE: If ALU overflows, Overflow=1	ac+i—ac	ac = # i = # CRY = x	result unchanged ALU CRY
NADI n.ac NOTE: If ALU overflows, Overflow=1	ac + n→ac	ac = # n = # (1-4) CRY = x	result unchanged ALU CRY
NBSAC	Search from (AC1) to Q tail for (AC1+AC3)=all Os (16-bit test)	AC1=E(1st Q data element) AC3=2#(word offset) (wsp)=mask word	E(last Q data element searched) unchanged unchanged
NBSAS	Search from (AC1) to Q tail for (AC1+AC3)=all 1s (16-bit test)	AC1 = E(1st Q data element) AC3 = 2#(word offset) (wsp) = mask word	E(last Q data element searched) unchanged unchanged
NBSE	Search from (AC1) to Q tail for (AC1+AC3)=(wsp) (16-bit test)	AC1=E(1st Q data element) AC3=2#(word offset) (wsp)=mask word	E(last Q data element searched) unchanged unchanged
NBSGE	Search from (AC1) to Q tail for (AC1+AC3)<=(wsp) (16-bit test)	AC1 = E(1st Q data element) AC3 = 2#(word offset) (wsp) = mask word	E(last Q data element searched) unchanged unchanged
NBSLE	Search from (AC1) to Q tail for (AC1+AC3)>=(wsp) (16-bit test)	AC1=E(1st Q data element) AC3=2#(word offset) (wsp)=mask word	E(last Q data element searched) unchanged unchanged
NBSNE	Search from (AC1) to Q tail for (AC1+AC3) ≠ (wsp) (16-bit test)	AC1=E(1st Q data element) AC3=2#(word offset) (wsp)=mask word	E(last Q data element searched) unchanged unchanged
NBSSC	Search from (AC1) to Q tail for (AC1+AC3)=some Os (16-bit test)	AC1=E(1st Q data element) AC3=2#(word offset) (wsp)=mask word	E(last Q data element searched) unchanged unchanged
NBSSS	Search from (AC1) to Q tail for (AC1+AC3)=some 1s (16-bit test)	AC1=E(1st Q data element) AC3=2#(word offset) (wsp)=mask word	E(last Q data element searched) unchanged unchanged
NCLID	CPU id—AC0&AC1&AC2	AC0=x AC1=x AC2=x	model number microcode revision memory size
NDIV acs,acd NOTE: If result <-32,768 or >32,767, or acs=0; Overflow=1, acd=unchanged	acd/acs—acd	acs = 2# acd = 2#	unchanged quotient
*NEG/c]/sh]/#] acs,acd[,skip] NOTE: If acs=0, CRY→CRY	-acs—acd	acs = # acd = x	unchanged 2#(acs)
NFSAC	Search from (AC1) to Q head for (AC1+AC3)=all Os (16-bit test)	AC1 = E(1st Q data element) AC3 = 2#(word offset) (wsp) = mask word	E(last Q data element searched) unchanged unchanged

Instruction Format	Action	Before (Location =)	After (Location =)
NFSAS	Search from (AC1) to Q head for (AC1+AC3)=all 1s (16-bit test)	AC1=E(1st Q data element) AC3=2#(word offset) (wsp)=mask word	E(last Q data element searched) unchanged unchanged
NFSE	Search from (AC1) to Q head for (AC1+AC3)=(wsp) (16-bit test)	AC1=E(1st Q data element) AC3=2#(word offset) (wsp)=mask word	E(last Q data element searched) unchanged unchanged
NFSGE	Search from (AC1) to Q head for (AC1+AC3) $<$ = (wsp) (16-bit test)	AC1=E(1st Q data element) AC3=2#(word offset) (wsp)=mask word	E(last Q data element searched) unchanged unchanged
NFSLE	Search from (AC1) to Q head for (AC1+AC3) $>$ = (wsp) (16-bit test)	AC1 = E(1st Q data element) AC3 = 2#(word offset) (wsp) = mask word	E(last Q data element searched) unchanged unchanged
NFSNE	Search from (AC1) to Q head for (AC1+AC3) ≠ (wsp) (16-bit test)	AC1 = E(1st Q data element) AC3 = 2#(word offset) (wsp) = mask word	E(last Q data element searched) unchanged unchanged
NFSSC	Search from (AC1) to Q head for (AC1+AC3)=some 0s (16-bit test)	AC1 = E(1st Q data element) AC3 = 2#(word offset) (wsp) = mask word	E(last Q data element searched) unchanged unchanged
NFSSS	Search from (AC1) to Q head for (AC1+AC3) = some 1s (16-bit test)	AC1 = E(1st Q data element) AC3 = 2#(word offset) (wsp) = mask word	E(last Q data element searched) unchanged unchanged
NLDAI i,ac	i→ac	i=2# ac=x	unchanged 2#
NMUL acs,acd NOTE: If result $<$ -32,768 or $>$ 32,767, Over- flow=1	acd*acs—acd	acs = 2# acd = 2#	unchanged result
NNEG acs,acd NOTE: If acs=100000 ₈ , Overflow=1	-acs—acd	acs = # acd = x CRY = x	unchanged result ALU CRY
NSALA i,ac	If i AND $\overline{ac} = 0 = skip$	i = # ac = #	unchanged unchanged
NSALM i,ac	If i AND $\overline{(ac)} = 0 = skip$	i = # (ac) = #	unchanged unchanged
NSANA i,ac	If i AND ac \neq 0 = skip	i=# ac=#	unchanged unchanged
NSANM i,ac	If i AND (ac) \neq 0 = skip	i = # (ac) = #	unchanged unchanged
NSBI n,ac NOTE: If ALU overflows, Overflow=1	ac-n→ac	ac=# n=# (1-4) CRY=x	result unchanged ALU CRY
NSUB acs.acd NOTE: If ALU overflows, Overflow=1	acd-acs—acd	acs = # acd = # CRY = x	unchanged result ALU CRY

Instruction Format	Action	Before (Location =)	After (Location =)
*NIO[f] device	/f/→BUSY, DONE flags	BUSY, DONE flags = x	[f]
ORFB	reference bits (AC1),(AC1) + 1(AC1) + 16 OR (AC2)—(AC2)	ACO = pageframe count AC1 = pageframe # AC2 = E(word string) referenced bits = x	count-1 #+16 E+1
PATU	Purges address translator	OVR = x $CRY = x$	unchanged unchanged
PBX NOTE: Popped PC must reference a BKPT instruction	-12 words—wide stack execute(ACO)	stack = stack popped PC = x AC0 = 16-bit instr. OVR = x CRY = x	-6 double words PC+1 unchanged ?
PIO acs,acd	command(acs)—device	acs = command acd = source	unchanged destination
*POP acs,acd NOTE: If acs is acd, 1 word is popped	stack—acs to acd	stack = stack acs = x acd = x sp = x fp = x	-n words(n = 1-4) 1st stack word last stack word sp-n unchanged
*POPB	stack—registers	stack = stack CRY = x PC = (POPB) AC3 = x AC2 = x AC1 = x AC0 = x	-5 words 1st word(0) (1-15) 2nd word 3rd word 4th word 5th word
*POPJ	stack—PC	stack = stack PC = x sp = x fp = x	-1 word stack word -1 unchanged
*PSH acs,acd NOTE: If acs is acd, I accumulator is pushed	acs to acd—stack	stack = stack sp = x fp = x acs = x acd = x	+ n words(n = 1-4) + n unchanged unchanged unchanged
*PSHJ [@]displacement[,index]	PC+1—stack E—PC	PC = PC stack = stack	E + (PC + 1)
*PSHR	PC+2—stack	stack = stack	+(PC+2)
*READS ac NOTE: READS ac = DIA ac,CPU	console switches—ac	ac=x ION flag=x	result unchanged
RRFB	0—referenced bits (AC1,AC1+1,AC1+16)	AC0=pageframe count AC1=pageframe # referenced bits=x	count-1 # + 16 0
*RSTR	stack —locations	stack = stack CRY = x PC = (RSTR) AC3 = x AC2 = x AC1 = x AC0 = x stack fault addr. = x sl = x fp = x sp = x	-9 words 1st word(0) (1-15) 2nd word 3rd word 4th word 5th word 6th word 7th word 8th word 9th word
*RTN	stack—locations	stack = stack CRY = x PC = (RTN) AC3 = x AC2 = x AC1 = x AC0 = x sp = x fp = x	-5 words 1st word(0) (1-15) 2nd word 3rd word 4th word 5th word fp-5 AC3(popped)

Instruction Format	Action	Before	After
		(Location =)	(Location =)
*SAVE i	5 + i—stack	i=#	unchanged
		stack = stack	+5+i
		1st word	ACO
		2nd word	AC1
		3rd word	AC2
		4th word	fp(before SAVE)
		5th word(0)	CRY
		(1-15)	AC3
		sp = x	+5+i
		AC3 = x	fp
		fp = x	+5
SAVZ	Eandaa.l.		
SAVZ	5 words—stack	stack = stack	+5
		1st word	AC0
		2nd word	AC1
		3rd word	AC2
		4th word	fp(before SAVE)
		5th word(0)	CRY
		(1-15)	
			AC3
		sp = x	+5
		AC3 = x	fp
		fp = x	sp(before push)
*SBI n,ac	ac-n-→ac	ac=#	result
522 M, uc	uc n—uc		
		n = #(1-4)	unchanged
SEX acs,acd	acs(16 bits)—acd(32 bits)	acs=#	unchanged
	(sign extended)	acd=x	result
*SGE acs,acd	-		
SGE acs,aca	If acs>=acd =skip	acs = 2#	unchanged
		acd=2#	unchanged
*SGT acs,acd	If acs>acd = skip	acs=2#	unchanged
,	" doos dod okip	acd=2#	<u> </u>
tour .			unchanged
*SKPt device	If $t = \text{true} = \text{skip}$	BUSY,DONE flags = x	unchanged
SMRF	New values—mod&ref bits	AC0=new values	unchanged
		AC1 = pageframe #	-
		· -	unchanged
		modified bit=x	updated
		referenced bit = x	updated
*SNB acs,acd	If $(E)bit = 1 = skip$	acs = word pointer	unchanged
NOTE: If acs is acd, word pointer=0		acd=word offset &	unchanged
		bit pointer	dichanged
		•	
		(E) = x	unchanged
SNOVR	If $OVR = 0 = skip$	OVR = x	unchanged
SPSR	ACO—PSR	AC0=x	_
~- ~	700-1011		unchanged
		OVR=x	updated
		CRY=x	unchanged
*STA ac,[@]displacement[,index]	ac—→(E)	ac = #	unchanged
, , ,		(E) = x	#
STAED 40	(••
STAFP ac	ac—wfp	ac=#	unchanged
		wfp = x	#
STASB ac	acwsb	ac = #	unchanged
CITE A CIT		wsb=x	#
STASL ac	ac—wsl	ac=#	unchanged
		wsl=x	#
STASP ac	20		
SIASI UL	ac→wsp	ac=#	unchanged
		wsp=x	#
	ac—(wsp)	ac=#	unchanged
STATS ac			
STATS ac	αc—(₩3ρ)	(wsn) = v	#
		(wsp) = x	#
STATS ac *STB acs,acd	ac(right)—√(E)	(wsp) = x acs = bp	# unchanged

Instruction Format	Action	Before (Location =)	After (Location =)
*STI fpac	fpac(E)#	fpac = fp# AC 1 = data indicator AC 2 = x AC 3 = bp	unchanged unchanged AC3 last bp + 1
*STIX NOTE: If E is not large enough, I—CRY	fpac(1-4)(E)#	fpac(1-4) = fp# AC1 = data indicator AC2 = x AC3 = bp CRY = x	unchanged unchanged AC3 last bp + 1 0
*SUB/c/[sh]/#] acs,acd[,skip] NOTE: If result >32,768, CRY→CRY	acd-acs—acd	acs = # acd = # CRY = x	unchanged result ALU CRY
*SZB acs,acd NOTE: If acs is acd, word pointer=0	If (E) bit $= 0 = skip$	acs = word pointer acd = word offset & bit pointer (E) = x	unchanged unchanged unchanged
*SZBO acs,acd NOTE: If acs is acd, word pointer=0	If (E)bit=0 = skip $1 \rightarrow (E)$ bit	acs = word pointer acd = word offset & bit pointer (E) = x	unchanged unchanged updated
VBP	bp ? valid reference If ACO segment #>= AC1 segment #=skip	AC0=bp AC1=segment #	unchanged unchanged
VWP	word pointer ? valid reference If ACO segment #>= AC1 segment #=skip	AC0 = word pointer AC1 = segment #	unchanged unchanged
WADC acs,acd NOTE: If ALU overflows, Overflow=1	acs + acd-→acd	acs = # acd = # CRY = x	unchanged result ALU CRY
WADD acs,acd NOTE: If ALU overflows, Overflow=1	acs + acd—acd	acs = # acd = # CRY = x	unchanged result ALU CRY
WADDI i,ac NOTE: If ALU overflows, Overflow=1	i+ac→ac	i=# ac=# CRY=x	unchanged result ALU CRY
WADI n.ac NOTE: If ALU overflows, Overflow=1	n+ac→ac	n=# (1-4) ac=# CRY=x	unchanged result ALU CRY
WANC acs,acd	acs AND acd—acd	acs = # acd = #	unchanged result
WAND acs,acd	acs AND acd—acd i AND ac—ac	acs = # acd = # i = #	unchanged result unchanged
WANDI i,ac WASH acs,acd	acd(shift)—acd	ac=# acd=#	result result
		acs=# for shift (+=left, -=right)	unchanged
WASHI i,ac	ac(shift)—ac	ac=# i=# for shift (+=left, -=right)	result unchanged
WBLM	S(E)→D(E)	AC1=no. words (2#) (+=asc., -=desc.) AC2=source E AC3=destination E	0 last E ± 1 last E ± 1
WBR displacement	PC + displacement → PC	PC=PC	PC+displ.
WBSAC	(32-bit test) see NBSAC		
WBSAS	(32-bit test) see NBSAS		

Instruction Format	Action	Before (Location =)	After (Location =)
WBSE	(32-bit test) see NBSE		
WBSGE	(32-bit test) see NBSGE		
WBSLE	(32-bit test) see NBSLE		
WBSNE	(32-bit test) see NBSNE		
WBSSC	(32-bit test) see NBSSC		
VBSSS	(32-bit test) see NBSSS		
WBTO acs,acd	1—(E)bit	acs=word pointer acd=word offset & bit pointer	unchanged
		(E)bit=x	unchanged 1
NBTZ acs, acd	0—(E)bit	acs=word pointer acd=word offset &	unchanged
		bit pointer (E) bit $= x$	unchanged O
VCLM acs,acd	acs>=L & acs<=H=skip	acs=2# acd=(L) 2# (L+1)=H 2#	unchanged unchanged unchanged
	If acs is acd	acs = 2 # $(CLM) + 1 = L$ $(CLM) + 2 = H$	unchanged unchanged unchanged
VCMP	string 1 ? string 2 $-1 \longrightarrow AC1 (1 < 2)$ $0 \longrightarrow AC1 (1 = 2)$	AC0 = str2 #bytes (# = asc, 2# = desc) AC1 = str1 #bytes	0 or unmoved bytes
	+1 →AC1 (1>2)	(# = asc, 2# = desc) AC2 = str2 bp AC3 = str1 bp	last bp $+$ 1 or failing byte last bp $+$ 1 or failing byte
VCMT	$S(E) \longrightarrow D(E)$	ACO = del.table addr. AC1 = # bytes (#=asc, 2#=desc)	unchanged O or unmoved bytes
	If byte = delimiter = halt instruction	AC2 = destination bp AC3 = source bp	last bp + 1 last bp + 1
VCMV	S(E)→D(E) CRY=relative length	ACO = dest. #bytes (#=asc, 2#=desc)	0
	1→S>D 0→S= <d< td=""><td>AC1 = source #bytes (# = asc, 2# = desc)</td><td>0 or unmoved bytes</td></d<>	AC1 = source #bytes (# = asc, 2# = desc)	0 or unmoved bytes
		AC2 = destination bp AC3 = source bp CRY = x	last bp + 1 last bp + 1 rel. length
VCOB acs,acd	acs(1's) +acd—acd	acs = # acd = 2#	unchanged result
VCOM acs,acd	acs—acd	acs = # acd = x	unchanged result
VCST	If (AC3) = delimiter = halt instruction	ACO = del. table addr. AC1 = #bytes (# = asc, 2# = desc) AC3 = bp	unchanged 0 or unmoved bytes
/CTR	C(E) D(E)	CRY=x	last bp + 1 or delimiter ?
CIR	S(E)→D(E) translates	ACO = trans.table bp AC1 = # bytes (2#) AC2 = destination bp AC3 = source bp	unchanged O last bp + 1 last bp + 1
	OR		·
	string1 ? string2 translates	AC0 = trans.table bp	unchanged
	translates -1→AC1 (1<2)	AC1=# bytes (#) AC2=destination bp	result code last bp + 1
	$0 \rightarrow AC1 (1 = 2) + 1 \rightarrow AC1 (1 > 2)$	AC3=source bp	last bp + 1

Instruction Format	Action	Before (Location =)	After (Location =)
WDIV acs,acd NOTE: If result <-2,147,483,648 or >2,147,483,647, or acs=0,	acd/acs—acd	acd=2# acs=2#	result(quotient) unchanged
then Overflow=1 and acs=unchanged			
WDIVS NOTE: If result <-2,147,483,648 or >2,147,483,647, or AC2=0, then Overflow=1 and AC0&AC1=unchanged	AC0&AC1/AC2→ AC1(quotient)& AC0(remainder)	AC0 = 2#(high-order) AC1 = 2#(low-order) AC2 = 2# (divisor)	remainder quotient unchanged
WDPOP	Restores CPU state	(32-33)page zero = context block	unchanged
WEDIT	See EDIT instruction The only changes for the WEDIT instruction subopcodes are: The value for j: If j=-; word is at (wsp+2+2*j) The value of k in the DDTK and DSTK subopcodes: If (k)=-, stack word=wsp+2+2*k If (k)=+, stack word=wfp+2+2*k		
WFFAD ac.fpac	absolute(fpac)—ac	fpac=fp#	unchanged
NOTE: If fpac <-2,147,483,648 or >2,147,483,647, then I—MOF and if fpac=+, ac=#; if fpac=-, ac=2#		ac=x FPSR(N,Z)	# or 2# unchanged
WFLAD acspac	ac—fpac	ac=2# fpac=x FPSR(N,Z)	unchanged fp#d updated
WFPOP NOTE: FPSR(12-15) are not set by WFPOP	stack—registers	stack = stack FPAC3 = x FPAC2 = x FPAC1 = x FPAC0 = x FPSR = x	-20 words 1st 4 2nd 4 3rd 4 4th 4 5th 4
WFPSH	registers—stack	stack = stack 1st 4 = x 2nd 4 = x 3rd 4 = x 4th 4 = x 5th 4 = x	+ 20 words FPSR FPAC0 FPAC1 FPAC2 FPAC3
WFSAC	(32-bit test) see NFSAC		,
WFSAS	(32-bit test) see NFSAS		
WFSE	(32-bit test) see NFSE		
WFSGE	(32-bit test) see NFSGE		
WFSLE	(32-bit test) see NFSLE		
WFSNE	(32-bit test) see NFSNE		
WFSSC	(32-bit test) see NFSSC		
WFSSS	(32-bit test) see NFSSS		
WHLV ac NOTE: WHLV rounds toward 0	ac/2—ac	ac=2#	result
WINC acs,acd	acs+1→acd	acs=#	unchanged
NOTE: If ALU overflows, Overflow=1		acd=x CRY=x	# + 1 ALU CRY
WIOR acs,acd	acs OR acd—acd	acs=# acd=#	unchanged result

Instruction Format	Action	Before (Location =)	After (Location =)
WIORI i,ac	ac OR i→ac	ac = # i = #	result unchanged
WLDAI i,ac	i→ac	i=# ac=x	unchanged #
WLDB acs,acd	(acs)—acd	acs=bp acd=x	unchanged byte
WLDI fpac	dec#(E)—fp#(fpac)	(E) = decimal # fpac = x AC1 = data indicator AC2 = x AC3 = bp FPSR(N,Z)	unchanged norm fp# unchanged AC3 last bp + 1 updated
WLDIX	dec#(E)→fp#(fpac0,1,2,3)	FPAC0=x FPAC1=x FPAC2=x FPAC3=x FPSR(N,Z) AC1=data indicator AC2=x AC3=bp	1st unit 2nd unit 3rd unit 4th unit ? unchanged AC3 last bp + 1
WLMP	(E)→map slots	AC0=#(1st slot #) AC1=#(# slots) AC2=E	last slot # 0 lastE+2
WLOB acs,acd	acs(# of 0s)+acd→acd	acs=# acd=2#	unchanged result
WLRB acs,acd NOTE: If acs is acd, then no count taken but 0→leading I	acs(# of 0s) + acd—acd O—lead 1(acs)	acs = # acd = 2#	new # result
WLSH acs,acd	acd(shifted)—acd (acs determines shift)	acs=2# (+=left, -=right) acd=#	unchanged result
WLSHI i,ac	ac(shifted)—ac (i determines shift)	i=2# (+=left, -=right) ac=#	unchanged result
WLSI n,ac	ac(shift left(n))—ac	n=#(1-4) ac=#	unchanged result
WLSN	(E) = (non-0 or 0, + or -) +1 \rightarrow AC1 = (+ non-0) -1 \rightarrow AC1 = (non-0) 0 \rightarrow AC1 = (+0) -2 \rightarrow AC1 = (-0)	AC1 = data type AC2 = x AC3 = bp (E) = decimal #	value code AC3 ? unchanged
WMESS	If (((E) XOR (ACO)) and (AC3))=0 Then ((AC1)→(E) and (E)→(AC1)) PC+2→PC Else (E)→(AC1) PC+1→PC.	(E) = S# AC0 = H&L AC1 = D# AC2 = E AC3 = mask	unch OR D# unch S# unch unch
WMOV acs,acd	acs—acd	acs = # acd = #	unchanged acs
WMOVR ac	ac(shift right 1)—ac	ac = word pointer	byte pointer
WMSP ac	sp+ac→sp	ac = 2# sp = #	unchanged result
WMUL acs,acd NOTE: If result <-2,147,483,648 or >2,147,483,647, Overflow=1	acd*acs—acd	acs=2# acd=2#	unchanged result
WMULS	AC1*AC2+AC0→ AC0&AC1	AC0=2# AC1=2# AC2=2#	result(high) result(low) unchanged
WNADI i,ac NOTE: If ALU overflows, Overflow=1	ac +i—ac	ac = # n = 2# CRY = x	result unchanged ALU CRY

Instruction Format	Action	Before (Location =)	After (Location =)
WNEG acs,acd NOTE: If acs=0, CRY→CRY, If ALU overflows, Overflow=1	-acs—acd	acs=# acd=x CRY=x	unchanged acs ALU CRY
WPOP acs,acd NOTE: If acs is acd, 1 double word is popped	stack→acs to acd	stack = stack acs = x acd = x wsp = x	-n(1-4) double words 1st stack double word last stack double word -2*n
WPOPB	stack → registers	stack = stack CRY = x PC = (WPOPB) AC3 = x AC2 = x AC1 = x AC0 = x OVK = x OVR = x IRES = x wsp = x	-6 double words 1st double word(0) 1st double word(1-31) 2nd double word 3rd double word 4th double word 5th double word 6th double word(0) (1) (2) -((17-31)*2+12)
WPOPJ	stack—PC	stack = stack PC = x sp = x fp = x	-1 double word top stack word -2 unchanged
WPSH acs,acd NOTE: If acs is acd, I accumulator is pushed	acs to acd—stack	stack = stack sp = x fp = x acs = x acd = x	+ 2*n(1-4) + 2*n unchanged unchanged unchanged
WRTN	stack →locations wfp →wsp stack →locations	stack = stack CRY = x PC = (WRSTR) AC3 = x AC2 = x AC1 = x AC0 = x OVK = x OVR = x stack fault addr. = x wsb = x wsp = x wfp = x stack = stack CRY = x PC = (WRTN) AC3 = x AC2 = x AC1 = x AC0 = x	-10 double words 1st double word(0) (1-31) 2nd double word 3rd double word 4th double word 6th double word (1) (16-31) 7th double word 8th double word 9th double word 10th double word -6 double word 1st double word 1st double word(0) (1-31) 2nd double word 3rd double word 4th double word 5th double word
WSALA i,ac WSALM i,ac	See NSALA See NSALM	OVK = x OVR = x IRES = x wsp = x wfp = x	6th double word(0) (1) (2) -((17-31)*2+12) AC3(popped)
WSANA i,ac	See NSANA		
WSANM i,ac	See NSANM		

Instruction Format	Action	Before (Location =)	After (Location =)
WSAVR i	5 double words—stack	stack = stack 1st double word = x 2nd double word = x 3rd double word = x 4th double word = x 5th double word(0) = x (1-31) = x AC3 = x wfp = x	+5 double words AC0 AC1 AC2 AC3 CRY PC wsp(before push) wsp(before push)
WSAVS i	5 double words—stack	wsp=x OVK=x OVR=x stack=stack 1st double word=x	+ (2*i) 0 unchanged +5 double words AC0
		2nd double word=x 3rd double word=x 4th double word=x 5th double word(0)=x (1-31)=x AC3=x wfp=x wsp=x OVK=x OVR=x	AC1 AC2 AC3 CRY PC wsp(before push) wsp(before push) + (2*i) 1 unchanged
WSBI n,ac NOTE: If ALU overflows, Overflow=1	ac-n→ac	ac = # n = #(1-4) CRY = x	result unchanged ALU CRY
WSEQ acs,acd NOTE: If acd is acs, acs is compared with 0	If acs=acd =skip	acs = # acd = #	unchanged unchanged
WSEQI i,ac	If i=ac =skip	ac = # i = #	unchanged unchanged
WSGE acs,acd	If $acs > = acd = skip$	acs=2# acd=2#	unchanged unchanged
WSGT acs, acd NOTE: If acd is acs, acs is compared with 0	If acs>acd =skip	acs=2# acd=2#	unchanged unchanged
WSGTI i,ac	If ac>i =skip	i=# ac=#	unchanged unchanged
WSKBO bit number	If bit $\#(ACO) = 1 = skip$	ACO=#	unchanged
WSKBZ bit number WSLE acs,acd	If bit $\#(ACO) = 0 = skip$ If $acs < = acd = skip$	AC0=# acs=2#	unchanged unchanged
NOTE: If acd is acs, acs is compared with 0 WSLEI i,ac	If ac<=i =skip	acd=2# ac=# i=#	unchanged unchanged
WSLT acs,acd NOTE: If acd is acs, acs is compared with 0	If acs < acd = skip	acs=2# acd=2#	unchanged unchanged unchanged
WSNB acs,acd-	If bit(acs∿) = 1 = skip	acs = word pointer acd = word offset & bit pointer	unchanged unchanged
WSNE acs,acd NOTE: If acd is acs, acs is compared with 0	If acs ≠ acd = skip	acs = 2# acd = 2#	unchanged unchanged
VSNEI i,ac	If ac≠i =skip	ac=# i=#	unchanged unchanged

Instruction Format	Action	Before (Location =)	After (Location =)
WSSVR i	6 double words—stack	stack = stack	+6 double words
	0→0VK	1st double word=x 2nd double word=x 3rd double word=x 4th double word=x	PSR ACO AC1 AC2
		5th double word=x 6th double word=x	WFP (previous) (0) CRY (1-31) AC3
		wfp = x $AC3 = x$ $wsp = wsp$	wsp wsp +(2*i)
		OVK=x OVR=x	0
WSSVS i	Same as WSSVR except 1—OVK	OVK = x	1
WSTB acs,acd	ac(right)—→(E)	acs = bp acd = byte (E) = x	unchanged unchanged byte
WSTI fpac	fpac→(E)#	fpac=fp# AC1=data indicator AC2=x AC3=bp	unchanged unchanged AC3 last bp + 1
WSTIX NOTE: If E is not large enough, $1 \rightarrow CRY$	fpac(1-4)—(E)#	CRY=x fpac(1-4)=fp# AC1=data indicator AC2=x AC3=bp CRY=x	? unchanged unchanged AC3 last bp + 1 ?
WSUB acs,acd NOTE: If ALU overflows, Overflow=1	acd-acs—acd	acs = # acd = #	unchanged result
WSZB acs,acd	If (E) $bit = 0 = skip$	acs=word pointer acd=word offset & bit pointer	unchanged unchanged
	W. F. L	(E) = x	unchanged
WSZBO acs,acd	If (E) bit $= 0$ = skip 1 \rightarrow (E) bit	acs=word pointer acd=word offset & bit pointer (E)=x	unchanged unchanged updated
WUGTI i,ac	If ac>i =skip	ac = # i = #	unchanged unchanged
WULEI i,ac	If ac<=i =skip	ac = # i = #	unchanged unchanged
WUSGE acs, acd NOTE: If acd is acs, acs is compared with 0	If $acs > = acd = skip$	acs = # acd = #	unchanged unchanged
WUSGT acs, acd NOTE: If acd is acs, acs is compared with 0	If acs>acd =skip	acs = # acd = #	unchanged unchanged
WXCH acs,acd	acs<>acd	acs=# acd=#	acd acs
WXOP acs,acd,operation #	wide return block—stack E→PC	stack = stack 1st double word = x 2nd double word = x 3rd double word = x 4th double word = x 5th double word = x 6th double word = x	+6 double words PSR ACO AC1 AC2 AC3 (0) CRY (1-31) (WXOP+2)
		AC2=x AC3=x (12-13)page zero	(acs in stack) (acd in stack)
		= (table) E = op#+(12-13)	unchanged unchanged
		page zero	anonungea

Instruction Format	Action	Before	After
		(Location =)	(Location =)
WXOR acs,acd	acs XOR acd—acd	acs = # acd = #	unchanged result
WXORI i,ac	ac XOR i—ac	ac=# i=#	result unchanged
XCALL [@]disp[,index[arg count]]	If E=valid, E→PC	PC=PC AC3=x OVR=x	E PC+3 0
*XCH acs,acd	acs<>acd	acs = # acd = #	acd acs
*XCT ac NOTE: If $ac = 1$ st word of 2-word instruction, (XCT) + $1 = 2$ nd word	execute(ac)	ac = instruction	unchanged
XFAMD fpac,[@]displacement[,index]	(E) + fpac→fpac	(E) = fp#d fpac = fp#d	unchanged fp#d
XFAMS fpac,[@]displacement[,index]	(E) + fpac→fpac	(E)=fp#s fpac=fp#s	unchanged fp#s
XFDMD fpac,[@]displacement[,index]	fpac/(E)—fpac	(E)=fp#d fpac=fp#d	unchanged fp#d
XFDMS fpac,[@]displacement[,index]	fpac/(E)—fpac	(E)=fp#s fpac=fp#s	unchanged fp#s
XFLDD fpac,[@]displacement[,index]	(E)—fpac	(E)=fp#d fpac=fp#d	unchanged fp#d
XFLDS fpac,[@]displacement[,index]	(E)—fpac	(E)=fp#s fpac=fp#s	unchanged fp#s
XFMMD fpac,[@]displacement[,index]	fpac*(E)—fpac	(E)=fp#d fpac=fp#d	unchanged fp#d
XFMMS fpac,[@]displacement[,index]	fpac*(E)—fpac	(E)=fp#s fpac=fp#s	unchanged fp#s
XFMSD fpac,[@]displacement[,index]	fpac-(E)—fpac	(E) = $fp#d$ fpac = fp#d	unchanged fp#d
XFSMS fpac,[@]displacement[,index]	fpac-(E)—fpac	(E)=fp#s fpac=fp#s	unchanged fp#s
XFSTD fpac,[@]displacement[,index]	fpac→(E)	fpac=fp#d (E)=x	unchanged fp#d
XFSTS fpac,[@]displacement[,index]	fpac—(E)	fpac=fp#s (E)=x	unchanged fp#s
XJMP [@]displacement[,index]	E⊸PC	PC=PC	E
XJSR [@]displacement[,index]	E→PC	PC = PC $AC3 = x$	E PC+2
XLDB ac,displacement[,index]	(E)—ac	ac=bp (E)=byte	byte unchanged
XLEF ac,[@]displacement[,index]	E→ac	ac=x	E
XLEFB ac, displacement [, index]	E-→ac	ac=x	E(byte)
XNADD ac,[@]displacement[,index] NOTE: If ALU overflows, Overflow=1	(E) +ac→ac	(E) = # ac = # CRY = x	unchanged result ALU CRY
XNADI n,[@]displacement[,index] NOTE: If ALU overflows, Overflow=1	n + (E) → (E)	(E) = # n = # (1-4) CRY = x	result(い): サナハ unchanged ALU CRY
XNDIV $ac_{\cdot}[@]$ displacement[,index] NOTE: If $(E) = 0$ or result $> 32,767$ or < -32,768, Overflow=1	ac/(E)—ac	(E) = # ac = #	unchanged result
XNDO ac,term offset,[@]disp[,index] NOTE: If ALU overflows, Overflow=1	(E)+1→(E) If (E)>ac then PC+1+ termination offset→PC (E)→ac	(E) = # ac = # CRY = x	unchanged (E) ALU CRY

Instruction Format	Action	Before (Location =)	After (Location =)	
XNDSZ [@]displacement[,index]	(E)-1→(E) If (E)=0 = skip	(E) = #		
XNISZ [@]displacement[,index]	$(E) + 1 \longrightarrow (E)$ If $(E) = 0 = skip$	(E) = #	(E) + 1	
XNLDA ac,[@]displacement[,index]	(E)—ac	(E) = # ac = x	unchanged (E)	
XNMUL ac./@/displacement/.index/ NOTE: If result > 32,767 or <-32,768, Over-flow=1	(E)*ac→ac	(E) = # ac = #	unchanged result	
XNSBI n,(@]displacement(,index) NOTE: If ALU overflows, Overflow=1	(E)-n—(E)	(E) = # n = # (1-4) CRY = x	#-n unchanged ALU CRY	
XNSTA ac,[@]displacement[,index]	ac—(E)	ac = # $(E) = x$	unchanged ac	
XNSUB ac,[@]displacement[,index] NOTE: If ALU overflows, Overflow=1	ac-(E)—ac	ac = # (E) = #	result unchanged	
XOP0 acs,acd,operation #	narrow return block—stack E—PC	stack = stack 1st word = x 2nd word = x 3rd word = x 4th word = x 5th word = x AC2 = x AC3 = x (44)page zero	+5 words AC0 AC1 AC2 AC3 (0) CRY (1-15) (XOP0+1) (acs in stack) (acd in stack)	
XOR acs,acd	acs XOR acd—acd	= (table) E = op# + (44) acs = #	unchanged unchanged unchanged	
XORI i,ac	ac XOR i—ac	acd=# ac=#	result ,	
VDEE (Old:l		i=#	unchanged	
XPEF [@]displacement[,index] XPEFB displacement[,index]	E—wide stack E—wide stack	stack = stack	+E	
XPSHJ [@]displacement[,index]	PC+3→wide stack E→PC	stack = stack PC = PC stack = stack	+ E(byte) E + (PC + 3)	
XSTB ac,displacement[,index]	ac—(E)	(E) = x ac = byte	byte unchanged	
XVCT [@]displacement	See instruction dictionary	25 2715	anonangou	
XWADD ac.[@]displacement[.index] NOTE: If ALU overflows, Overflow=1	ac+(E)→ac	(E) = # ac = # CRY = x	unchanged result ALU CRY	
XWADI n.[@]displacement[.index] NOTE: If ALU overflows, Overflow=1	(E) + n→(E)	(E) = # n = # (1-4) CRY=x	result unchanged ALU CRY	
XWDIV ac,[@]displacement[,index] NOTE: If quotient <-2,147,483,648 or >+2,147,483,647, or (E)=0; Overflow=1, and ac=unchanged	ac/(E)—ac(quotient)	(E) = # ac = # CRY = x	unchanged quotient ALU CRY	
XWDO ac,term offset,[@]disp[,index] NOTE: If ALU overflows, Overflow=1	(E) + 1→(E) If (E)>ac then PC+1+ termination offset→PC If (E) <=ac then PC+1→PC (E)→ac	(E) = # ac = # CRY=x	#-1 (E)+1 ALU CRY	
XWDSZ [@]displacement[,index]	(E)-1=(E) If (E)=0 = skip	(E) = #	#-1	
XWISZ [@]displacement[,index]	(E) + 1 = (E) If $(E) = 0 = skip$	(E) = #	#+1	
XWLDA ac,[@]displacement[,index]	(E)—ac	(E) = #	unchanged	

Instruction Format	Action	Before (Location =)	After (Location =)
XWMUL ac.[@]displacement[,index] NOTE: If result >-2,147,483,648 or <+2,147,483,647, Overflow=1	(E)*ac→ac	(E) = # ac = #	unchanged result
XWSBI n,[@]displacement[.index] NOTE: If ALU overflows, Overflow=1	(E)-n→(E)	(E) = # n = # (1-4) CRY = x	#-n unchanged ALU CRY
XWSTA ac,[@]displacement[,index]	ac→(E)	ac = # (E) = x	unchanged ac
XWSUB ac,[@]displacement[.index] NOTE: If ALU overflows, Overflow=1	ac-(E)—ac	ac=# (E)=# CRY=x	result unchanged ALU CRY
ZEX acs,acd	acs—acd (zero extended)	acs=#(16-bit) acd=x	unchanged acs#(32-bit)

Appendix B Anomalies

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 77777₈ 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 77777₈. This means that a C/350 program counter 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 is the same; that is, it contains the address of the floating-point instruction that caused the 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 is AC1, and the meaning of the PC in the return block are identical to the results obtained on the C/350.

Appendix C **ASCII Codes**

KEY DECIMAL OCTAL HEX SYMBOL MINEMONIC	KEY DECIMAL OCTAL HEX SYMBOL	KEY DECIMAL OCTAL HEX SYMBOL	KEY DECIMAL OCTAL HEX SYMBOL
0 000 00 1 A NUL	32 040 20 SPACE	65 101 41 A	97 141 61 a
1 001 01 A SOH	33 041 21 !	66 102 42 B	98 142 62 b
2 002 02 B STX	34 042 22 (QUOTE)	67 103 43 C	99 143 63 C
3 003 03 1 C ETX	35 043 23 #	68 104 44 D	100 144 64 d
4 004 04 1 D EOT	36 044 24 \$	69 105 45 E	101 145 65 e
5 005 05 FE ENQ	37 045 25 %	70 106 46 F	102 146 66 f
6 006 06 F ACK	38 046 26 &	71 107 47 G	103 147 67 g
7 007 07 † G BEL	39 047 27 (APOS)	72 110 48 H	104 150 68 h
8 010 08 H BS BACKSPACE	40 050 28 (73 111 49 I	105 151 69 i
9 011 09 1 TAB	41 051 29)	74 112 4A J	106 152 6A j
10 012 OA J NEW LINE	42 052 2A *	75 113 4B K	107 153 68 k
11 013 OB K VT	43 053 2B +	76 114 4C L	108 154 6C I
12 014 OC L FORM	44 054 2C (COMMA)	77 115 4D M	109 155 6D m
13 015 OD M CARRIAGE RETURN	45 055 2D -	78 116 4E N	110 156 6E n
14 016 OE N SO	46 056 2E (PERIOD)	79 117 4F O	111 157 6F o
15 017 OF 10 SI	47 057 2F /	80 120 50 P	112 160 70 P
16 020 10 P DLE	48 060 30 0	81 121 51 Q	113 161 71 q
17 021 11 Q DC1	49 061 31 1	82 122 52 R	114 162 72 r
18 022 12 R DC2	50 062 32 2	83 123 53 S	115 163 73 s
19 023 13 +S DC3	51 063 33 3	84 124 54 T	116 164 74 t
20 024 14 T DC4	52 064 34 4	85 125 55 U	117 165 75 u
21 025 15 TU NAK	53 065 35 5	86 126 56 V	118 166 76 v
22 026 16 V SYN	54 066 36 6	87 127 57 W	119 167 77 w
23 027 17 W ETB	55 067 37 7	88 130 58 X	120 170 78 x
24 030 18 X CAN	56 070 38 8	89 131 59 Y	121 171 79 y
25 031 19 Y EM	57 071 39 9	90 132 5A Z	122 172 7A Z
26 032 1A +Z SUB	58 072 3A :		
27 033 1B ESC ESCAPE	59 073 3B ;	91 133 58 [123 173 7B [
28 034 1C 1\ FS	60 074 3C <	92 134 5C \	124 174 7C I
29 035 1D 1 GS	61 075 3D -	93 135 5D l	125 175 7D]
30 036 1E † RS	62 076 3E <	94 136 5E 1 OR	126 176 7E TILDE
31 037 1F + US	63 077 3F ?	95 137 5F OR	127 177 7F DEL
	64 100 40 @	96 140 60 (GRAVE)	

Appendix D Powers of 2 Table

```
2^n
                              1/2<sup>n</sup>
                       0
                              1.0
              2
                              0.5
                              0.25
              8
                              0.125
                              0.062
                                     5
              16
                                     25
             32
                              0.031
             64
                              0.015
                                     625
            128
                              0.007
                                     812
                                     906
            256
                      8
                              0.003
                                           25
            512
                              0.001
            024
                      10
                              0.000
                                     976
                                           562 5
            048
                      11
                              0.000
                                     488
                                           281
                                                25
            096
                     12
                                     244
                              0.000
                                           140 625
            192
                      13
                              0.000
                                     122
                                          070
                                               312
       16
            384
                     14
                              0.000
                                     061
                                          035
                                               156
       32
                     15
            768
                              0.000
                                     030
                                          517 578
                                                     125
       65
            536
                     16
                              0.000
                                     015
                                          258
                                               789
                                                     062
                                                          5
       131
            072
                              0.000
                                          629
      262
            144
                     18
                              0.000
                                     003
                                                     265
      524
            288
                     19
                              0.000
                                     001
                                          907
                                               348
                                                     632
                                                          812
      048
            576
                     20
                              0.000
                                     000
                                          953 674
                                                     316
                                                                25
      097
                     21
            152
                              0.000
                                     000
                                          476 837
                                                     158
                                                          203
                                                                125
      194
            304
                     22
                              0.000
                                     000
                                          238
                                               418
                                                     579
                                                          101
      388
                                     000
                              0.000
                                          119
                                               209
                                                     289
                                                                781
                                                                     25
                                                          550
  16
      777
            216
                     24
                              0.000
                                     000
                                          059
                                               604
                                                     644
                                                          775
                                                                390
                                                                     625
 33
      554
                     25
            432
                                     000
                                               802
                              0.000
                                          029
                                                     322
                                                                695
                                                          387
                                                                     312
 67
      108
            864
                     26
                              0.000
                                     000
                                          014
                                               901
                                                     161
                                                          193
                                                                847
                                                                     656
134
      217
            728
                     27
                              0.000
                                     000
                                          007
                                               450
                                                     580
                                                          596
                                                                923
                                                                     828
268
      435
                     28
                              0.000
                                     000
                                          003
                                                     290
                                               725
                                                          298
                                                                461
                                                                     914
                                                                          062
536
      870
            912
                     29
                              0.000
                                     000
                                          001
                                               862
                                                     645
                                                          149
                                                                230
                                                                    957
                                                                          031
073
      741
                     30
                              0.000
                                     000
                                                                               625
                                          000
                                               931
                                                                615
                                                     322
                                                          574
                                                                     478
                                                                          515
147
      483
            648
                     31
                              0.000
                                     000
                                          000
                                               465
                                                     661
                                                          287
                                                                307
                                                                     739
                                                                          257
                                                                               812
294
      967
                              0.000
                                     000
                                          000
                                               232
                                                    830
                                                          643
                                                                653
                                                                    869
                                                                          628
                                                                               906
                                                                                     25
589
      934
            592
                     33
                              0.000 000 000
                                               116
                                                    415 321
                                                               826 934 814
                                                                               453
                                                                                     125
```

Figure D.1 Powers of 2 Table

Appendix E 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 E.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

Table E.1 Protection Violation Fault Codes

Stack Faults

Table E.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 E.2 Wide Stack Fault Codes

Decimal/ASCII Faults

Table E.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 Retur	ned in AC1		
Narrow	Wide	Faulting Instruction	Meaning
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	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	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	WLSN, WLDI, LSN, LDI LDIX, WLDIX EDIT, WEDIT	Sign code is invalid for this data type
000007	100007	WLSN, WLDI, WLDIX, LSN LDI, LDIX	Invalid digit

Table E.3 Decimal and ASCII fault codes

Appendix F 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 "f" following a page entry indicate "and the following pages".	AC1 with fault code, loading 5-19f, 9-3 value identifying fault, setting 8-13 AC1, fault code in 5-17ff, 9-3, E-2
15-bit displacement 1-11 16-bit data, edit decimal and alphanumeric 9-8, 10-35f fixed-point logical format 2-12 two's complement 2-2 instructions, executing 9-1 integer, convert to 10-19 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 fixed-point logical format 2-12 fixed-point two's complement 2-2 processor instruction, equivalent 9-7 programs, supporting 4-1 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	relative address mode 1-12 with base address of DCT, loading 7-12 AC3 relative address mode 1-12 Access fault, privileged 5-12 flag, execute 8-5 read 8-5 valid page 8-4 write 8-5 instructions, double-word stack 4-5 privileges destination segment 5-12 request 8-7 restrictions, memory 1-6 validation 8-7 violation, (read, write, and execute) 8-13f Access, device 7-1 execute 1-9 ignoring page protocol 8-6f operand 1-12 page 8-4, 8-7 read 1-9 segment 8-2
@ flag 9-5f A	type of page 8-4 type of segment 8-2 valid
Aborting floating-point division 3-11 instruction execution 2-23, 8-2 memory reference instruction 8-2 nonprivileged fault 5-14, 8-13f subroutine call 5-11f Absolute address 1-10, 1-12f value 3-3, 10-42 AC (see accumulator) AC0 with address of fault instruction, setting 8-13 fault, loading 5-20 AC0, load PSR in 2-9f, 10-82 store PSR from 2-9, 10-110	page 8-4 read 8-5 segment 8-2 validate I/O 8-2 write 1-9 Accessing destination segment 5-9f device control table 7-12 double word in a stack 4-4 memory 1-8, 8-2 page table 8-4 to read, write, or execute 8-4 physical page 8-2 protection mechanism 1-17

Accumulator (also see AC0, AC1, etc.)	store 9-8, 10-110
bit	storing byte address in 2-23
instructions, wide skip on 2-23	transfers between device and 7-1
set to one, wide skip on 2-8, 2-23, 10-150	wide
set to zero, wide skip on 2-8, 2-23, 10-150	add memory word to 2-4, 10-85, 10-172
contents, executing 5-2	fix from floating-point 3-3f, 3-11, 9-9, 10-131
equal to immediate, wide skip if 10-149	float from 3-3f
greater than immediate,	float from fixed-point 3-3f, 9-9, 10-131
wide skip if 2-8, 10-150	load 2-3, 10-87, 10-174
wide unsigned skip if 2-8, 10-156	skip on all bits set in 2-8, 10-145
in	skip on any bit set in 2-8, 10-146
double word addressed by WSP, store 2-3, 4-5	store 2-3, 10-88, 10-175
stack pointer contents, store 10-111	Accumulators,
WFP, store 4-4, 8-13f, 10-110	exchange 2-3, 10-160
WSB, store 4-4, 10-111	
WSL, store 4-4, 10-111	pop floating-point status and 10-51f
WSP, store 4-4, 10-111	
less than or equal to immediate,	multiple 9-8, 9-11, 10-101, 10-142
wide skip if 2-8, 10-151	popping 4-1
wide unsigned skip if 2-8, 10-157	push floating point status and 10,526
not equal to immediate, wide skip if 2-8, 10-152	floating-point status and 10-52f
relative addressing, C/350 9-1	multiple 9-8, 9-11, 10-102f
to memory word, narrow add 2-4	pushing 4-1
with	restoring 5-4
double word addressed by WSP, load 4-5	storing 5-7 wide
double word, load 10-67	
WFP, load 4-4, 10-67	pop multiple 2-3, 4-1, 4-5, 4-8, 9-8, 9-11, 10-142
WSB, load 4-4, 10-67	push 2-3, 4-5, 4-8, 9-8, 10-144
WSL, load 4-4, 10-67	Accuracy of result, increasing 3-4
WSP, load 4-4, 10-67	ADD 2.4, 2.6, 2.8, 10.2f
Accumulator,	ADD 2-4, 2-6, 2-8, 10-3ff Add
C/350	
fixed-point 9-1	accumulator to memory word, narrow 2-4
floating-point 9-1	and move, block 9-8, 10-8f
double word addressed by WSP load 2-3	complement,
extended load 9-8, 10-38	wide 2-4, 10-117
extended store 9-8, 10-41	with optional skip 2-4, 2-6, 2-8, 10-2f
fix to 10-45	double (FPAC to FPAC) 3-6, 10-42
fixed-point 1-2	double (memory to FPAC) 3-6, 9-9, 10-42, 10-71, 10-161
float from 10-47	
load 9-8, 10-66	immediate 2-4, 10-5f
loading	immediate,
byte in 2-20	extended 2-4, 10-5
physical address into 7-3	narrow 2-4, 10-77, 10-92, 10-166
narrow	narrow extended 2-4, 10-91
add memory word to 2-4, 10-77, 10-166	wide 2-4, 10-85, 10-117f, 10-173
load 2-3, 9-8, 10-79, 10-168	
skip on all bits set in 2-8, 10-97	
skip on any bit set in 2-8, 10-98	
store 2-3 0-8 10-80 10-160	

instructions, fixed-point 2-22	Address space,
memory word to accumulator,	C/350 9-4ff
narrow 2-4, 10-77, 10-166	logical 8-1, 8-10
wide 2-4, 10-85, 10-172	memory 1-1
single (FPAC to FPAC) 3-6, 10-43	valid 2-22
single (memory to FPAC) 3-6, 9-9, 10-42f, 10-71,	Address to device, sending physical 7-2
10-161	Address to original fault, return 8-13
to DI 10-20	Address translation,
Add to	device 7-1, 7-6
P 10-20f	memory 8-2f, 8-6
SI 10-21	Address translator, purging 8-11, 10-100
Add with	Address wraparound 1-4f, 5-1, B-1
narrow immediate, wide 2-4, 10-142	Address,
optional skip 2-4, 2-6, 2-8, 10-3ff	6-bit device 7-4
wide immediate, wide 2-4, 10-117	absolute 1-12
Add,	base 1-11, 7-12
decimal 10-20	C/350 effective 9-2, 9-5
narrow 2-4, 10-91	calculating an effective 2-21f
wide 2-4, 10-117	checking range of the logical 8-6
with optional skip 2-6	direct 1-15
ADDI 2-4, 10-5	effective 1-10f, 1-16, 2-20f, 5-2, 5-4f, 5-10, 7-6, 7-11.
Adding one to the intermediate mantissa 3-6	8-2
Addition instructions,	logical 8-6
fixed-point 2-4	relative 1-11
floating-point 3-6	extended load effective 9-10, 10-39
Addition, floating-point 3-5f	forming byte 9-5f
Additional bytes, inserting 2-21	forming physical 7-2
Address	indirect 1-10, 1-15
boundary, two Kbyte 8-3	logical 8-6
causing fault, decimal instruction 5-18	relative 1-11
field,	intermediate logical 1-11
device 7-4	load
word 1-14f	effective 2-12ff, 2-21, 8-3, 9-10, 10-70, 10-76, 10-165
in accumulator, storing byte 2-23	effective byte 1-13, 2-21, 10-76, 10-166
into accumulator, loading physical 7-3 mnemonic, device 7-4	physical 7-2
	logical 5-1, 8-2ff, F-1
mode AC and PC relative 1-12 Address of	logical address to physical 8-1
block to transfer, starting 7-2	most significant bits translating logical 7-2
• •	narrow return 5-22
DCT, loading AC2 with base 7-12	one Mbyte or less logical 8-3
decimal fault instruction 5-19 fault instruction,	one Mbyte to 512 Mbytes logical 8-3
at time of fault 5-14ff	one-level page table logical word 8-6
	PC return 4-6
determining 5-16ff	physical 1-7, 1-11, 7-2, 8-1ff
setting AC0 with 8-13	page 8-6
fault, loading AC0 with 5-20	page table 8-3
first entry in page table 8-2ff	
first word for I/O transfer 7-2	
floating-point instruction 3-11	
sub-block to transfer, starting 7-2	
WEDIT fault instruction 5-18	
Address parameter, vector stack fault 7-9	
Address ranges 1-10, 1-12, 8-6, 9-10 Address reference fault code, inward F-1	
ADDIESS (EIECENCE 1200 CODE IDW270 F.)	

push 2-21, 4-5, 4-8, 10-81, 10-171	Alphanumeric 16-bit data, edit decimal and 9-8, 10-35f
byte 2-21	Alphanumeric, wide edit of 5-5
effective 2-21	Altering normal program flow 5-1
effective byte 2-20f, 4-5f, 4-8, 10-81, 10-171	Always, skip 10-54
return 5-6, 9-10f, 10-103	ANC 2-13, 10-6
return 5-2, 5-7, 5-13, 5-15f, 5-20	AND 2-6, 2-12ff, 10-6ff
skip on valid byte or word 2-22	immediate 2-13, 10-8
store return 5-7, 7-7	immediate, wide 2-13, 10-118
translating logical 1-7, 7-2f, 8-2, 8-4	with complemented source 2-13, 10-6, 10-118
two-level page table logical word 8-6	with optional skip 2-6, 2-12ff, 10-6ff
Addressing	AND, wide 2-13, 10-118
another	ANDI 2-13, 10-8
	Anomalies B-1
page table page 8-6	
segment 1-9	Another segment 1-9, 4-2, 4-4, 5-6
chains, indirect 1-11	ANY flag 3-10f, 5-15
memory for I/O data transfer 7-2	Appending guard digit 3-4
next instruction 5-1	Argument count 5-7
protection violation, indirect 1-11	Arguments,
range 1-10, 8-3, 8-6	save without 9-10f, 10-106f
scheme F-1	transferring call 5-10ff
violation, indirect 8-13f	Arithmetic
Addressing,	example, decimal 2-23
absolute 1-10, 1-13	instructions,
byte and word formats for 1-9	decimal 2-22
C/350 9-1ff	fixed-point 2-4
device control table 7-12	floating-point 3-6
direct 8-6	shift
effective 1-13	instructions, wide 2-6
entry in the vector table 7-12	with narrow immediate, wide 10-119
indirect 1-8, 1-11, 8-6, 9-5f	shift, wide 10-118f
mapped memory 1-5	Arithmetic,
memory 1-5, 1-8	binary coded decimal 2-1
n variable in PC relative 1-11	expanding to 32-bit fixed-point 9-3f
relative 1-10f, 1-13	Array,
	expanding data areas for large 9-3
unmapped memory 1-5	gate 5-10f
ADI 2-4, 10-5f	ASCII
Algorithms, floating-point numerical B-2	
Aligning mantissa 3-4f	characters 2-15f
All bits set in	codes C-1
accumulator,	data
narrow skip on 10-97	fault E-1
wide skip on 10-145	fault code E-2
memory location,	data,
narrow skip on 10-97f	fault 5-16
wide skip on 10-146	invalid format 5-13
Alphabetical characters, move j 2-21	type 2 or 3 fault return block for 5-18f
Alphabetics, move 10-27	valid type 2-23
	fault code 5-17
	or decimal fault code 5-14, 5-16f, E-2
	Assignments and format, map register 7-2
	Attempting division by zero 2-8, 5-14f

addressing, format C/350 9-5ff format wide 1-15 and set bit to one, skip on zero 9-5f, 9-8, 10-115f wide skip on zero 9-8, 10-156 operand 1-15 pointer 1-15, 9-5ff set in accumulator, narrow skip on any 10-146 in memory location, narrow skip on any 10-146 to one, wide skip on accumulator 2-23, 10-150 to zero, wide skip on accumulator 2-23, 10-150 three of SBR, setting 7-2 Bit to one, set 9-5ff, 10-10 skip on zero bit and set 9-5f, 9-8, 10-115f wide set 9-8, 10-121f wide set 9-8, 10-122 two of SBR, setting 7-1ff zero of fault code in AC1 9-3 narrow stack pointer 5-22f narrow stack pointer 5-22f narrow stack pointer 5-22f narrow stack pointer 5-22f narrow stack pointer 5-22f narrow stack pointer 5-22f narrow stack pointer 5-22f NOR referenced 8-11, 10-103 store modified and referenced 8-11, 10-109 table of referenced 8-11, 10-109 table of referenced 8-11, 10-103 store modified and referenced 8-11, 10-109 table of referenced 8-11, 10-103 store modified and referenced 8-11, 10-109 table of referenced 8-11, 10-109 table of referenced 8-11, 10-103 store modified and referenced 8-11, 10-109 table of referenced 8-11, 10-109 table of referenced 8-11, 10-103 store modified and referenced 8-11, 10-109 table of referenced 8-11, 10-103 store modified and referenced 8-11, 10-109 table of referenced 8-11, 10-103 store modified and referenced 8-11, 10-109 table of referenced 8-11, 10-103 store modified and referenced 8-11, 10-109 table of referenced 8-11, 10-103 store modified and referenced 8-11, 10-103 store modified and referenced 8-11, 10-109 table of referenced and modified 8-1 wide count 9-8, 10-126 BKPT 2-9, 4-5, 4-8, 5-5f, 5-23, 9-10f, 10-100, 10-102 wide pop 2-9, 4-5, 4-8, 5-5f, 5-23, 9-10f, 10-100, 10-102 wide pop 2-9, 4-5, 4-8, 5-5f, 5-23, 9-10f, 10-100, 10-102 wide pop 2-9, 4-5, 4-8, 5-5f, 5-23, 9-10f, 10-100, 10-102 wide pop 2-9, 4-5, 4-8, 5-5f, 5-23, 9-10f, 10-100 move, wide stack return 4-5 move 9-8, 10-10 move, wide stack return 8-13 floating-point fault return 5-16 narrow return 9-2 pop 5-23, 9-10f, 10-102	В	wide stack limit 5-20f
ink 6-2 search queue and skip, narrow 5-3, 6-5, 10-92f wide 5-3, 6-5, 10-92f wide 5-3, 6-5, 10-120f BAM 9-8, 10-8f Base address 1-11, 7-12 level interrupt processing 5-8, 7-9ff, 10-144f of vector table 7-12 register, segment 5-11, 8-16, 8-6 wide stack 8-13f Base, initializing wide stack 8-13f Base, initializing wide stack 4-2, 8-10, 8-13f loading wide stack 5-11, 5-13 wide stack 4-2, 4-7, 5-11, 5-13, 5-20, 7-9, 7-11, 8-10 BCD 2-1, 2-15ff, 2-22 BI field 1-14 Binary coded decimal arithmetic 2-1 conversion instructions, floating-point 3-3 operations 2-1 point location 3-2 Bit addressing, format C/350 9-5ff format wide 1-15 and set bit to one, skip on zero 9-8, 10-156 set one, skip on any 10-98 wide skip on any 10-146 in memory location, narrow skip on any 10-146 in memory location, narrow skip on any 10-146 in one, wide skip on accumulator 2-23, 10-150 three of SBR, setting 7-2 Bit to one, set 9-5ff, 10-10 skip on zero bit and set 9-8, 10-12f wide skip on accumulator 2-23, 10-150 three of SBR, setting 7-2 bit of zero, set 9-5ff, 10-10 skip on zero bit and set 9-8, 10-12f wide skip on zero b		wide stack pointer 5-20f
search queue and skip, narrow 5.3, 6-5, 10-92f wide 5-3, 6-5, 10-120f BAM 9-8, 10-8f Baze address 1-11, 7-12 level interrupt processing 5-8, 7-9ff, 10-144f of vector table 7-12 register, segment 5-11, 8-16, 8-16, 8-13f loading wide stack 8-13f Base, initializing wide stack 4-2, 8-10, 8-13f loading wide stack 5-11, 5-13 wide stack 4-2f, 4-7, 5-11, 5-13, 5-20, 7-9, 7-11, 8-10 BCD 2-1, 2-15ff, 2-22 Bif field 1-14 Binary coded decimal arithmetic 2-1 conversion instructions, floating-point 3-3 operations 2-1 point location 3-2 Bit addressing, format C/350 9-5ff format wide 1-15 and set bit to one, skip on zero 9-8, 10-115f wide skip on zero 9-8, 10-115f wide skip on amy 10-46 in memory location, narrow skip on amy 10-146 in one, wide skip on any 10-146 in memory location, narrow skip on amy 10-146 in one, wide skip on any 10-146 in memory location, narrow skip on amy 10-146 in one, wide skip on any 10-146 in one, wide skip on any 10-146 in one, wide skip on are 9-8, 10-12f wide set 9-8, 10-12f wide set 9-8, 10-12f wide skip on zero bit and set 9-8, 10-12f wide skip on		Bit,
narrow 5-3, 6-5, 10-120f BAM 9-8, 10-8f Base address 1-11, 7-12 tevel interrupt processing 5-8, 7-9ff, 10-144f of vector table 7-12 register, segment 5-11, 8-1f, 8-6 wide stack 8-13f Base, initializing wide stack 4-2, 8-10, 8-13f loading wide stack 4-2f, 4-7, 5-11, 5-13, 5-20, 7-9, 7-11, 8-10 BCD 2-1, 2-15ff, 2-22 Bl field 1-14 Binary coded decimal arithmetic 2-1 conversion instructions, floating-point 3-3 operations 2-1 Binary coded decimal arithmetic 2-1 conversion instructions, floating-point 3-3 operations 2-1 Bin format C/350 9-5ff format wide 1-15 and set bit to one, skip on zero 9-8, 10-115f wide skip on zero 9-8, 10-115f wide skip on any 10-98 wide skip on any 10-146 in memory location, narrow skip on any 10-146 in memory location, narrow skip on any 10-146 in memory location, narrow skip on any 10-146 in zero, wide skip on accumulator 2-23, 10-150 three of SBR, setting 7-2 Bit to one, set 9-5ff, 10-10 skip on zero bit and set 9-8, 10-115f wide skip on zero bit and set 9-8, 10-121f wide skip on zero bit and set 9-8, 10-121f wide skip on zero bit and set 9-8, 10-121f wide skip on zero bit and set 9-8, 10-126 wide set 9-8, 10-121f wide skip on zero bit and set 9-8, 10-126 for zero, set 9-5ff, 10-10f wide skip on zero bit and set 9-8, 10-126 for zero, set 9-5ff, 10-10f wide skip on zero bit and set 9-8, 10-126 for zero, set 9-5ff, 10-10f wide skip on zero bit and set 9-8, 10-127 marrow stack limit 5-22f narrow stack pointer 5-22f		checking validity 5-11
wide 5-3, 6-5, 10-120f BAM 9-8, 10-8f Base address 1-11, 7-12 level interrupt processing 5-8, 7-9ff, 10-144f of vector table 7-12 register, segment 5-11, 8-16, 8-6 wide stack 8-13f Base, initializing wide stack 4-2, 8-10, 8-13f loading wide stack 5-11, 5-13 wide stack 4-2f, 4-7, 5-11, 5-13, 5-20, 7-9, 7-11, 8-10 BCD 2-1, 2-15ff, 2-22 BIf field 1-14 Binary coded decimal arithmetic 2-1 conversion instructions, floating-point 3-3 operations 2-1 Bit addressing, format C/530 9-5ff format wide 1-15 and set bit to one, skip on zero 9-5f, 9-8, 10-115f wide skip on zero 9-8, 10-156 operand 1-15 operand 1-15 operand 1-15 pointer 1-15, 9-5ff set in accumulator, narrow skip on any 10-146 in memory location, narrow skip on any 10-146 in accumulator, narrow skip on any 10-146 in one, wide skip on accumulator 2-23, 10-150 to zero, wide skip on accumulator 2-23, 10-150 to zero, wide skip on accumulator 2-23, 10-150 to zero, set 9-5ff, 10-10f wide set 9-8, 10-121f wide skip on zero bit and set 9-8, 10-126 wide skip on accumulator 3-2 true of SBR, setting 7-16f zero of fault code in AC1 9-3 narrow stack limit 5-22f narrow stack klimit 5-22f narrow stack klimit 5-22f narrow stack bointer 5-22f marrow stack pointer 5-22f		indirect 9-5f
BAM 9-8, 10-8f Base address 1-11, 7-12 level interrupt processing 5-8, 7-9ff, 10-144f of vector table 7-12 register, segment 5-11, 8-1f, 8-6 wide stack 8-13f Base, initializing wide stack 4-2, 8-10, 8-13f loading wide stack 4-2f, 4-7, 5-11, 5-13, 5-20, 7-9, 7-11, 8-10 BDC 2-1, 2-15ff, 2-22 BI field 1-14 Binary coded decimal arithmetic 2-1 conversion instructions, floating-point 3-3 operations 2-1 point location 3-2 Bit addressing, format C/350 9-5ff format wide 1-15 and set bit to one, skip on zero 9-8, 10-115f wide skip on zero 9-8, 10-115f wide skip on any 10-98 wide skip on any 10-98 wide skip on any 10-146 in memory location, narrow skip on any 10-18 in accumulator, narrow skip on any 10-18 wide skip on any 10-18 in accumulator, narrow skip on any 10-98 wide skip on any 10-146 in memory location, narrow skip on any 10-146 in memory location, narrow skip on any 10-18 wide skip on arcumulator 2-23, 10-150 three of SBR, setting 7-2 Bit to one, set 9-5ff, 10-10 skip on zero bit and set 9-8, 10-115f wide skip on zero bit and set 9-8, 10-121f wide skip on zero bit and set 9-8, 10-121f wide skip on zero bit and set 9-8, 10-126 wide set 9-8, 10-121f wide skip on zero bit and set 9-8, 10-156 to zero, set 9-5ff, 10-10f swide set 9-8, 10-122 two of SBR, setting 7-1ff zero of fault code in AC1 9-3 narrow stack limit 5-22f narrow stack klimit 5-22f narrow stack klimit 5-22f narrow stack klimit 5-22f narrow stack pointer 5-22f		locate lead 10-80
Base address 1-11, 7-12 level interrupt processing 5-8, 7-9ff, 10-144f of vector table 7-12 register, segment 5-11, 8-1f, 8-6 wide stack 8-13f Base, initializing wide stack 4-2, 8-10, 8-13f loading wide stack 8-11, 5-13 wide stack 9-21, 2-15ff, 2-22 llat fidely and stack stac		mask 7-12
address 1-11, 7-12 level interrupt processing 5-8, 7-9ff, 10-144f of vector table 7-12 register, segment 5-11, 8-1f, 8-6 wide stack 4-18, 4-7, 5-11, 5-13 wide stack 4-2, 8-10, 8-13f loading wide stack 4-2, 8-10, 8-13f loading wide stack 4-15, 1-5, 13 wide stack 4-21, 4-7, 5-11, 5-13, 5-20, 7-9, 7-11, 8-10 BCD 2-1, 2-15ff, 2-22 BI field 1-14 Binary coded decimal arithmetic 2-1 conversion instructions, floating-point 3-3 operations 2-1 point location 3-2 Bit addressing, format C/350 9-5ff format wide 1-15 and set bit to one, skip on zero 9-8, 10-1156 wide skip on any 10-186 in accumulator, narrow skip on any 10-186 in accumulator, narrow skip on any 10-186 in memory location, narrow skip on any 10-186 in memory location, narrow skip on any 10-186 in memory location, narrow skip on any 10-186 in one, wide skip on accumulator 2-23, 10-150 to zero, set 9-5ff, 10-10 skip on zero bit and set 9-8, 10-122 two of SBR, setting 7-1ff zero of fault code in AC1 9-3 narrow stack pointer 5-22f and row stack pointer 5-22f are stack limit 5-22f narrow stack pointer 5-22f		sign 3-2
level interrupt processing 5-8, 7-9ff, 10-144f of vector table 7-12 register, segment 5-11, 8-1f, 8-6 wide stack 8-1, 8-6 wide stack 8-1, 8-6 wide stack 8-13f loading wide stack 4-2, 8-10, 8-13f loading wide stack 8-2, 11, 5-13 s. 5-20, 7-9, 7-11, 8-10 BCD 2-1, 2-15ff, 2-22 Bl field 1-14 loading wide stack 8-11, 5-13 s. 5-20, 7-9, 7-11, 8-10 loading wide stack 8-2f, 4-7, 5-11, 5-13, 5-20, 7-9, 7-11, 8-10 loading wide stack 8-11, 5-13 s. 5-20, 7-9, 7-11, 8-10 loading wide stack 8-11, 10-197 wide stack 8-11, 10-103 store modified and referenced 8-11, 10-109 loa		skip on
wide catched to the first property of the content o		nonzero 9-5ff, 10-109
register, segment 5-11, 8-1f, 8-6 wide stack 8-13f Base, initializing wide stack 4-2, 8-10, 8-13f loading wide stack 5-11, 5-13 wide stack 4-2f, 4-7, 5-11, 5-13, 5-20, 7-9, 7-11, 8-10 BCD 2-1, 2-15ff, 2-22 BIT goild 1-14 Binary coded decimal arithmetic 2-1 conversion instructions, floating-point 3-3 operations 2-1 point location 3-2 Bit addressing, format C/350 9-5ff format wide 1-15 and set bit to one, skip on zero 9-5f, 9-8, 10-115f wide skip on any 10-146 in memory location, narrow skip on any 10-186 vin example 1-15, 9-5ff set in accumulator, narrow skip on any 10-186 in memory location, narrow skip on any 10-186 to one, wide skip on accumulator 2-23, 10-150 three of SBR, setting 7-12 Bit to one, set 9-5ff, 10-10 skip on zero bit and set 9-5f, 9-8, 10-115f wide set 9-8, 10-121f wide set 9-8, 10-122 two of SBR, setting 7-12 two of SBR, setting 7-16 zero of fault code in AC1 9-3 narrow stack limit 5-22f narrow stack pointer 5-22f in cotate and reset lead 10-138 locate lead 10-137 skip on zero lead 10-137 skip on zero skip on any 10-97 wide skip on all 10-97 wide skip on all 10-146 Bits, count 9-8, 10-16 displacement (sign-extending to 31) 1-11 ignoring page access protocol 8-6 load modified and referenced 8-11, 10-70 OR referenced 8-11, 10-90 referenced 8-11, 10-90 reset referenced 8-11, 10-109 reset r		zero 9-8
segment 5-11, 8-16, 8-6 wide stack 8-13f Base, initializing wide stack 4-2, 8-10, 8-13f loading wide stack 4-2, 8-10, 8-13f loading wide stack 4-2, 7, 5-11, 5-13, 5-20, 7-9, 7-11, 8-10 BED 2-1, 2-15ff, 2-22 BI field 1-14 Binary coded decimal arithmetic 2-1 conversion instructions, floating-point 3-3 operations 2-1 point location 3-2 Bit addressing, format C/350 9-5ff format wide 1-15 and set bit to one, skip on zero 9-5f, 9-8, 10-115f wide skip on zero 9-5f, 9-8, 10-156 operand 1-15 pointer 1-15, 9-5ff set in accumulator, narrow skip on any 10-98 wide skip on any 10-98 wide skip on any 10-146 in memory location, narrow skip on any 10-146 in memory location, narrow skip on any 10-146 to one, wide skip on accumulator 2-23, 10-150 three of SBR, setting 7-2 Bit to one, set 9-5ff, 10-10 skip on zero bit and set 9-8, 10-115f wide set 9-8, 10-121f roover, set 9-5ff, 10-10 skip on zero bit and set 9-8, 10-156 to zero, set 9-5ff, 10-10 skip on zero bit and set 9-8, 10-156 to zero, set 9-5ff, 10-10 skip on zero bit and set 9-8, 10-156 to zero, set 9-5ff, 10-10 skip on zero bit and set 9-8, 10-156 to zero, set 9-5ff, 10-10 skip on zero bit and set 9-8, 10-156 to zero, set 9-5ff, 10-10f side set 9-8, 10-121f wide set 9-8, 10-121f wide set 9-8, 10-121f wide set 9-8, 10-121f side set 9-8, 10-121f wide set 9-8, 10-121f side set 9-8, 10-121f wide set 9-8, 10-121f side set 10-145 set tin accumulator, sarrow skip on all 10-97 side set in accumulator, sarrow skip on all 10-145 set tin accumulator, sarrow skip on all 10-146 set tim accumulator, sarrow skip on all 10-146 set tim accumulator, sarrow skip on all 10-146 set tim accumulator, sarrow skip on all 10-146 s		wide
skip on zero 9-8, 10-156 Base, initializing wide stack 4-2, 8-10, 8-13f loading wide stack 4-2, 8-10, 8-13f loading wide stack 4-2, 8-10, 15-13 wide stack 4-2, 1-47, 5-11, 5-13, 5-20, 7-9, 7-11, 8-10 BCD 2-1, 2-15ff, 2-22 BIT field 1-14 Binary coded decimal arithmetic 2-1 conversion instructions, floating-point 3-3 operations 2-1 point location 3-2 Bit addressing, format C/350 9-5ff format wide 1-15 and set bit to one, skip on zero 9-8, 10-115f wide skip on are 9-8, 10-115f wide skip on are 9-8, 10-115f set in accumulator, narrow skip on any 10-98 wide skip on any 10-146 in memory location, narrow skip on any 10-98 wide skip on any 10-146 to one, wide skip on accumulator 2-23, 10-150 three of SBR, setting 7-2 Bit to one, set 9-5ff, 10-10 skip on zero bit and set 9-5f, 9-8, 10-115f wide set 9-8, 10-121f wide set 9-8, 10-121f wide set 9-8, 10-122 two of SBR, setting 7-12 two of SBR, setting 7-12 two of SBR, setting 7-16r zero of fault code in AC1 9-3 narrow stack pointer 5-22f fault return 8-13 return 4-1, 4-5, 9-3 return 4-1, 9-10f, E-2		locate and reset lead 10-138
Base, initializing wide stack 4-2, 8-10, 8-13f loading wide stack 5-11, 5-13 wide stack 4-2f, 4-7, 5-11, 5-13, 5-20, 7-9, 7-11, 8-10 BIGD 2-1, 2-151ff, 2-22 BI field 1-14 Innoversion instructions, floating-point 3-3 operations 2-1 point location 3-2 Bit addressing, format C/350 9-5ff format wide 1-15 and set bit to one, skip on zero 9-5f, 9-8, 10-115f wide skip on zero 9-8, 10-156 operand 1-15 pointer 1-15, 9-5ff set in accumulator, narrow skip on any 10-98 wide skip on any 10-146 in memory location, narrow skip on any 10-146 in one, wide skip on accumulator 2-23, 10-150 three of SBR, setting 7-2 Bit to one, set 9-5ff, 10-10 skip on zero bit and set 9-8, 10-115f wide set 9-8, 10-121 wide skip on zero bit and set 9-8, 10-156 to zero, wide skip on accumulator 2-22, 10-150 troop of SBR, setting 7-15 wide set 9-8, 10-121 troop of SBR, setting 7-15 core, set 9-5ff, 10-10 skip on zero bit and set 9-8, 10-156 to zero, set 9-5ff, 10-10f wide set 9-8, 10-121 troop of fault code in AC1 9-3 narrow stack limit 5-22f narrow stack pointer 5-22f serious and stack stack return 4-1, 4-5, 9-3 return 4-1, 4-5, 9-3 return 4-1, 4-5, 9-3 return 4-1, 9-10, fine 2-2 serious and stack stack return 4-2 serious and stack stack return 4-2 serious and stack stack return 8-13 feutrum 4-1, 4-5, 9-3 return 4-1, 9-10, fine 2-2 serious and stack limit 5-22f serious stack pointer 5-22f serious and stack limit 5-22f serious stack pointer 5-22f serious and stack limit 5-22f serious stack pointer 5-22f serious and stack limit 5-22f serious stack pointer 5-22f serious and stack limit 5-22f serious stack pointer 5-22f serious and stack limit 5-22f serious stack pointer 5-22f serious and stack po		locate lead 10-137
initializing wide stack 4-2, 8-10, 8-13f loading wide stack 5-11, 5-13 wide stack 4-2, 4-7, 5-11, 5-13, 5-20, 7-9, 7-11, 8-10 BCD 2-1, 2-15ff, 2-22 Blf field 1-14 Binary coded decimal arithmetic 2-1 conversion instructions, floating-point 3-3 operations 2-1 point location 3-2 Bit addressing, format C/350 9-5ff format wide 1-15 and set bit to one, skip on zero 9-5f, 9-8, 10-115f wide skip on zero 9-8, 10-156 operand 1-15 pointer 1-15, 9-5ff set in accumulator, narrow skip on any 10-146 in memory location, narrow skip on any 10-146 in memory location, narrow skip on any 10-186 wide skip on any 10-146 in memory location, narrow skip on any 10-146 in one, wide skip on accumulator 2-23, 10-150 three of SBR, setting 7-2 Bit to one, set 9-5ff, 10-10 skip on zero bit and set 9-5f, 9-8, 10-115f wide set 9-8, 10-121f wide skip on zero bit and set 9-8, 10-156 to zero, set 9-5ff, 10-10 skip on zero bit and set 9-8, 10-156 to zero, set 9-5ff, 10-10 skip on zero bit and set 9-8, 10-156 to zero, set 9-5ff, 10-10 skip on zero bit and set 9-8, 10-156 to zero, set 9-5ff, 10-10 skip on zero bit and set 9-8, 10-156 to zero, set 9-5ff, 10-10 skip on zero bit and set 9-8, 10-156 to zero, set 9-5ff, 10-10 skip on zero bit and set 9-8, 10-156 to zero, set 9-5ff, 10-10 skip on zero bit and set 9-8, 10-156 to zero, set 9-5ff, 10-10 skip on zero bit and set 9-8, 10-156 to zero, set 9-5ff, 10-10 skip on zero bit and set 9-8, 10-156 to zero, set 9-5ff, 10-10 skip on zero bit and set 9-8, 10-156 to zero, set 9-5ff, 10-10 sylve the set of sample that the set of the s		skip on zero 9-8, 10-156
loading wide stack 4-2f, 4-7, 5-11, 5-13 wide stack 4-2f, 4-7, 5-11, 5-13, 5-20, 7-9, 7-11, 8-10 BCD 2-1, 2-15ff, 2-22 BI field 1-14 Bitnary coded decimal arithmetic 2-1 conversion instructions, floating-point 3-3 operations 2-1 point location 3-2 Bit addressing, format C/350 9-5ff format wide 1-15 and set bit to one, skip on zero 9-5f, 9-8, 10-115f wide skip on zero 9-8, 10-156 operand 1-15 pointer 1-15, 9-5ff set in accumulator, narrow skip on any 10-98 wide skip on any 10-146 in memory location, narrow skip on accumulator 2-23, 10-150 to zero, wide skip on accumulator 2-23, 10-150 to zero, wide skip on accumulator 2-23, 10-150 to zero, wide skip on accumulator 2-23, 10-150 to zero, set 9-5ff, 10-10 skip on zero bit and set 9-8, 10-12f wide set 9-8,	· ·	
wide stack 4-2f, 4-7, 5-11, 5-13, 5-20, 7-9, 7-11, 8-10 BCD 2-1, 2-15ff, 2-22 BI field 1-14 Binary coded decimal arithmetic 2-1 conversion instructions, floating-point 3-3 operations 2-1 point location 3-2 Bit addressing, format C/350 9-5ff format wide 1-15 and set bit to one, skip on zero 9-5f, 9-8, 10-115f wide skip on zero 9-95f, 9-8, 10-156 operand 1-15 pointer 1-15, 9-5ff set in accumulator, narrow skip on any 10-98 wide skip on any 10-146 in memory location, narrow skip on any 10-146 in memory location, narrow skip on any no-98 wide skip on any no-98 wide skip on any 10-146 in one, wide skip on accumulator 2-23, 10-150 to zero, wide skip on accumulator 2-23, 10-150 to zero, wide skip on accumulator 2-23, 10-150 wide set 9-8, 10-121f wide skip on zero bit and set 9-8, 10-156 to zero, set 9-5ff, 10-10 skip on zero bit and set 9-8, 10-156 to zero, set 9-5ff, 10-10 skip on zero bit and set 9-8, 10-156 to zero, set 9-5ff, 10-10 skip on zero bit and set 9-8, 10-156 to zero, set 9-5ff, 10-10 skip on zero bit and set 9-8, 10-156 to zero, set 9-5ff, 10-10 skip on zero bit and set 9-8, 10-156 to zero, set 9-5ff, 10-10 skip on zero bit and set 9-8, 10-156 to zero, set 9-5ff, 10-10 skip on zero bit and set 9-8, 10-156 to zero, set 9-5ff, 10-10 skip on zero bit and set 9-8, 10-156 to zero, set 9-5ff, 10-10 skip on zero bit and set 9-8, 10-156 to zero, set 9-5ff, 10-10 skip on zero bit and set 9-8, 10-156 to zero, set 9-5ff, 10-10 skip on zero bit and set 9-8, 10-156 to zero, set 9-5ff, 10-10 skip on zero bit and set 9-8, 10-156 to zero, set 9-5ff, 10-10 skip on zero bit and set 9-8, 10-156 to zero, set 9-5ff, 10-10 skip on zero bit and set 9-8, 10-156 to zero, set 9-5ff, 10-10 skip on zero bit and set 9-8, 10-156 to zero, set 9-5ff, 10-10 skip on zero bit and set 9-8, 10-156 to zero, set 9-5ff, 10-10 skip on zero bit and set 9-8, 10-156 to zero, set 9-5ff, 10-10 skip on zero bit and set 9-8, 10-156 to zero, set 9-5ff, 10-10 skip on zero zero, set 9-5ff, 10-10 skip on zero zero, set 9-5ff, 10-10 skip on zero zero, set		accumulator,
wide stack 4-2f, 4-7, 5-11, 5-13, 5-20, 7-9, 7-11, 8-10 BCD 2-1, 2-15ff, 2-22 BI field 1-14 Binary coded decimal arithmetic 2-1 conversion instructions, floating-point 3-3 operations 2-1 point location 3-2 Bit addressing, format C/350 9-5ff format wide 1-15 and set bit to one, skip on zero 9-5f, 9-8, 10-115f wide skip on zero 9-8f, 10-156 operand 1-15 pointer 1-15, 9-5ff set in accumulator, narrow skip on any 10-98 wide skip on any 10-146 in memory location, narrow skip on any 10-18 Bit to one, set 9-5ff, 10-10 skip on zero bit and set 9-5f, 9-8, 10-115f wide skip on zero bit and set 9-8, 10-156 to zero, set 9-5ff, 10-10 skip on zero bit an		narrow skip on all 10-97
BIG field 1-14 Binary coded decimal arithmetic 2-1 conversion instructions, floating-point 3-3 operations 2-1 point location 3-2 Bit addressing, format C/350 9-5ff format wide 1-15 and set bit to one, skip on zero 9-5f, 9-8, 10-115f wide skip on ary 10-146 set in accumulator, narrow skip on any 10-146 to one, wide skip on any 10-146 to one, wide skip on any 10-146 to one, wide skip on accumulator 2-23, 10-150 to zero, wide skip on accumulator 2-23, 10-150 to zero, wide skip on accumulator 2-23, 10-150 to set 9-5ff, 10-10 skip on zero bit and set 9-8, 10-115f wide set 9-8, 10-121f wide skip on zero bit and set 9-8, 10-156 to zero, set 9-5ff, 10-10 skip on zero bit and set 9-8, 10-156 to zero, set 9-5ff, 10-10 skip on zero bit and set 9-8, 10-156 to zero, set 9-5ff, 10-10f wide set 9-8, 10-121f wide set 9-8, 10-122f wide set 9-8, 10-122f set of SBR, setting 7-1 set referenced s-11, 10-109 table of referenced and modified 8-1 wide count 9-8, 10-126 BKPT 2-9, 4-5, 4-8, 5-5f, 10-9 BLM 9-8, 10-10 Block add and move 9-8, 10-8f and execute, pop 2-9, 4-5, 4-8, 5-5f, 5-23, 9-10f, 10-100, 10-102 wide pop 2-9, 4-5, 4-8, 5-8f, 5-14, 5-21, 9-10f, 10-104 in a stack, accessing doubte word 4-4 instructions, wide stack return 4-5 move, wide 2-3, 9-8, 10-119f to transfer, starting address of 7-2 Block. C/350 return 9-2 data 4-4 fault return 8-13 floating-point fault return 5-16 narrow return 9-2 by 5-23, 9-10f, 10-102 popping return 4-1, 4-5 wide return		
Bil field 1-14 Binary coded decimal arithmetic 2-1 conversion instructions, floating-point 3-3 operations 2-1 point location 3-2 Bit addressing, format C/350 9-5ff format wide 1-15 and set bit to one, skip on zero 9-5f, 9-8, 10-115f wide skip on zero 9-5f, 9-8, 10-115f wide skip on zero 9-8, 10-156 operand 1-15 pointer 1-15, 9-5ff set in accumulator, narrow skip on any 10-98 wide skip on any 10-146 in memory location, narrow skip on any 10-146 to one, wide skip on any 10-146 to one, set 9-5ff, 10-10 skip on zero bit and set 9-8, 10-121f wide set 9-8, 10-121f zero of SBR, setting 7-1ff zero of SBR, setting 7-1ff zero of fault code in AC1 9-3 narrow stack limit 5-22f narrow stack pointer 5-22f return 4-4, 9-10f, E-2	· · · · · · · · · · · · · · · · · · ·	
Binary coded decimal arithmetic 2-1 conversion instructions, floating-point 3-3 operations 2-1 point location 3-2 Bit addressing, format C/350 9-5ff format wide 1-15 and set bit to one, skip on zero 9-5f, 9-8, 10-115f wide skip on zero 9-8, 10-156 operand 1-15 pointer 1-15, 9-5ff set in accumulator, narrow skip on any 10-98 wide skip on any 10-146 in memory location, narrow skip on any 10-98 wide skip on any 10-146 to one, wide skip on accumulator 2-23, 10-150 to zero, wide skip on accumulator 2-23, 10-150 to set 9-5ff, 10-10 skip on zero bit and set 9-5f, 9-8, 10-115f wide set 9-8, 10-121f wide skip on zero bit and set 9-8, 10-156 to zero, set 9-5ff, 10-10 skip on zero bit and set 9-8, 10-156 to zero, set 9-5ff, 10-10f wide set 9-8, 10-121f wide skip on any 10-156 to zero, set 9-5ff, 10-10f wide set 9-8, 10-122f two of SBR, setting 7-1ff zero of fault code in AC1 9-3 narrow stack limit 5-22f narrow stack pointer 5-22f arrow stack pointer 5-22f set referenced 8-11, 10-77 OR referenced 8-11, 10-109 table of referenced 8-11, 10-109 table of referenced 8-11, 10-109 table of referenced and modified and referenced 8-11, 10-109 table of referenced 8-11, 10-109 table of referenced 8-11, 10-109 table of referenced and modified and referenced 8-11, 10-109 table of referenced 8-10-1010 store of referenced 8-11, 10-101 in one of set of refer	BI field 1-14	narrow skip on all 10-97f
coded decimal arithmetic 2-1 conversion instructions, floating-point 3-3 operations 2-1 point location 3-2 Bit addressing, format C/350 9-5ff format wide 1-15 and set bit to one, skip on zero 9-5f, 9-8, 10-115f wide skip on zero 9-8, 10-156 operand 1-15 pointer 1-15, 9-5ff set in accumulator, narrow skip on any 10-146 in memory location, narrow skip on any 10-146 to one, wide skip on accumulator 2-23, 10-150 to zero, wide skip on accumulator 2-23, 10-150 three of SBR, setting 7-2 Bit to one, set 9-5ff, 10-10 skip on zero bit and set 9-8, 10-121f wide set 9-8, 10-121f swide set 9-8, 10-121f wide set 9-8, 10-121f wide set 9-8, 10-121f wide set 9-8, 10-121f wide set 9-8, 10-121f swide set 9-8, 10-121f wide set 9-8, 10-121f wide set 9-8, 10-121f wide set 9-8, 10-121f swide set 9-8, 10-121f wide set 9-8, 10-121f swide set 9-8, 10-121f swide set 9-8, 10-121f wide skip on zero bit and set 9-8, 10-156 to zero, set 9-5ff, 10-10 skip on zero bit and set 9-8, 10-156 to zero, set 9-5ff, 10-10 skip on zero bit and set 9-8, 10-156 to zero, set 9-5ff, 10-10 skip on zero bit and set 9-8, 10-156 to zero, set 9-5ff, 10-10 skip on zero bit and set 9-8, 10-156 to zero, set 9-5ff, 10-10f wide set 9-8, 10-121f wide skip on zero bit and set 9-8, 10-156 to zero, set 9-5ff, 10-102 popping return 4-1, 4-5 wide return 8-13 floating-point fault return 8-13 floating-point fault return 8-13 floating-point fault return 8-13 floating-point fault return 8-13 protection violation return 8-13f pushing fault return 8-13 return 4-4, 9-10f, E-2	Binary	•
conversion instructions, floating-point 3-3 operations 2-1 point location 3-2 Bit addressing, format C/350 9-5ff format wide 1-15 and set bit to one, skip on zero 9-5f, 9-8, 10-115f wide skip on zero 9-8, 10-156 operand 1-15 pointer 1-15, 9-5ff set in accumulator, narrow skip on any 10-146 in memory location, narrow skip on any 10-146 to one, wide skip on accumulator 2-23, 10-150 to zero, wide skip on accumulator 2-23, 10-150 to zero, wide skip on accumulator 2-23, 10-150 to zero, wide skip on accumulator 2-23, 10-150 to zero bit and set 9-5f, 9-8, 10-115f wide set 9-8, 10-121f wide set 9-8, 10-121f wide set 9-8, 10-122 two of SBR, setting 7-1ff zero of fault code in AC1 9-3 narrow stack pointer 5-22f anarrow stack pointer 5-22f set of the fact of the set of	coded decimal arithmetic 2-1	
operations 2-1 point location 3-2 Bit addressing, format C/350 9-5ff format wide 1-15 and set bit to one, skip on zero 9-5f, 9-8, 10-115f wide skip on zero 9-8, 10-156 operand 1-15 pointer 1-15, 9-5ff set in accumulator, narrow skip on any 10-146 in memory location, narrow skip on any 10-146 to one, wide skip on accumulator 2-23, 10-150 three of SBR, setting 7-2 Bit to one, set 9-5ff, 10-10 skip on zero bit and set 9-5f, 9-8, 10-115f wide set 9-8, 10-121f wide set 9-8, 10-121f wide set 9-8, 10-121f wide set 9-8, 10-121f wide set 9-8, 10-122 two of SBR, setting 7-1ff zero of fault code in AC1 9-3 narrow stack jointer 5-22f narrow stack pointer 5-22f anarrow stack pointer 5-22f anarow stack pointer 5-22f displacement (sign-extending to 31) 1-11 ignoring page access protocol 8-6 load modified and referenced 8-11, 10-77 OR referenced 8-11, 10-109 rest referenced 8-11, 10-109 rest referenced 8-11, 10-109 rest referenced 8-11, 10-109 table of referenced 8-12, 10-109 wide count 9-8, 10-126 BKPT 2-9, 4-5, 4-8, 5-5f, 5-23, 9-10f, 10-100 move, wide 2-3, 9-8, 10-119f to transfer, s	conversion instructions, floating-point 3-3	
point location 3-2 Bit addressing, format C/350 9-5ff format wide 1-15 and set bit to one, skip on zero 9-5f, 9-8, 10-115f wide skip on zero 9-8, 10-156 operand 1-15 pointer 1-15, 9-5ff set in accumulator, narrow skip on any 10-98 wide skip on any 10-146 in memory location, narrow skip on any 10-146 to one, wide skip on accumulator 2-23, 10-150 to zero, wide skip on zero bit and set 9-5f, 9-8, 10-115f wide set 9-8, 10-121f wide skip on zero bit and set 9-8, 10-120 two of SBR, setting 7-1 Bit to one, set 9-5ff, 10-10f wide set 9-8, 10-121 wide skip on zero bit and set 9-8, 10-156 to zero, set 9-8, 10-121 for wide skip on zero bit and set 9-8, 10-156 to zero, set 9-8, 10-121 for zero of fault code in AC1 9-3 narrow stack pointer 5-22f anarrow stack pointer 5-22f anarrow stack pointer 5-22f serious difficid and referenced 8-11, 10-103 for ferenced 8-11, 10-103 for store modified and referenced 8-11, 10-103 for store modified and referenced 8-11, 10-103 for store modified and referenced 8-11, 10-109 table of referenced 8-11, 10-109 for Exert referenced 8-11, 10-103 for store modified and referenced 8-11, 10-109 for Exert referenced 8-11, 10-103 for store modified and referenced 8-11, 10-109 for Exert referenced 8-11, 10-103 for store modified and referenced 8-11, 10-103 for store modified and referenced 8-11, 10-109 for Exert referenced 8-11, 10-103 for store modified and referenced 8-11, 10-103 for store modified and referenced 8-11, 10-103 for store modified and referenced 8-11, 10-103 for store modifie		
addressing, format C/350 9-5ff format wide 1-15 and set bit to one, skip on zero 9-5f, 9-8, 10-115f wide skip on zero 9-8, 10-156 operand 1-15 pointer 1-15, 9-5ff set in accumulator, narrow skip on any 10-146 in memory location, narrow skip on any 10-146 in one, wide skip on ary 10-146 to one, wide skip on avourulator 2-23, 10-150 three of SBR, setting 7-2 Bit to one, set 9-5ff, 10-10 skip on zero bit and set 9-8, 10-115f wide set 9-8, 10-121f wide set 9-8, 10-121f wide set 9-8, 10-122 two of SBR, setting 7-1ff zero of fault code in AC1 9-3 narrow stack pointer 5-22f narrow stack pointer 5-22f narrow stack pointer 5-22f narrow stack pointer 5-22f in addressing, OR referenced 8-11, 10-109 store eferenced 8-11, 10-109 table of referenced 8-10, 10-10 bide oun 1-16 to end odified and	point location 3-2	
addressing, format C/350 9-5ff format wide 1-15 and set bit to one, skip on zero 9-5f, 9-8, 10-115f wide skip on zero 9-8, 10-156 operand 1-15 pointer 1-15, 9-5ff set in accumulator, narrow skip on any 10-186 wide skip on any 10-146 in memory location, narrow skip on any 10-186 wide skip on any 10-146 to one, wide skip on accumulator 2-23, 10-150 to zero, wide skip on accumulator 2-23, 10-150 to zero, wide skip on accumulator 2-23, 10-150 to three of SBR, setting 7-2 Bit to one, set 9-5ff, 10-10 skip on zero bit and set 9-8, 10-115f wide set 9-8, 10-121f wide skip on zero bit and set 9-8, 10-156 to zero, set 9-5ff, 10-10f wide set 9-8, 10-122 two of SBR, setting 7-1ff zero of fault code in AC1 9-3 narrow stack limit 5-22f narrow stack pointer 5-22f narrow stack pointer 5-22f OR referenced 8-11, 10-199 reset referenced 8-11, 10-109 table of referenced and modified 8-1 wide count 9-8, 10-126 BKPT 2-9, 4-5, 4-8, 5-5f, 5-23, 9-10f, 10-100, 10-102 wide pop 2-9, 4-5, 4-8, 5-5f, 5-23, 9-10f, 10-100, 10-102 wide pop 2-9, 4-5, 4-8, 5-5f, 5-23, 9-10f, 10-100, 10-102 wide pop 2-9, 4-5, 4-8, 5-5f, 5-23, 9-10f, 10-100, 10-102 wide pop 2-9, 4-5, 4-8, 5-5f, 5-23, 9-10f, 10-100, 10-102 wide pop 2-9, 4-5, 4-8, 5-5f, 5-23, 9-10f, 10-100, 10-102 wide pop 2-9, 4-5, 4-8, 5-5f, 5-23, 9-10f, 10-100, 10-102 wide pop 2-9, 4-5, 4-8, 5-5f, 5-23, 9-10f, 10-100, 10-102 wide pop 2-9, 4-5, 4-8, 5-5f, 5-23, 9-10f, 10-100 move, wide 2-3, 9-8, 10-119f to transfer, starting address of 7-2 Block, C/350 return 8-13 floating-point fault return 8-13 floating-point fault return 8-13 floating-point fault return 8-13 return 4-1, 4-5 wide count 9-8, 10-126 wide pop 2-9, 4-5, 4-8, 5-5f, 5-23, 9	Bit	
format C/350 9-5ff format wide 1-15 and set bit to one, skip on zero 9-5f, 9-8, 10-115f wide skip on zero 9-8, 10-156 operand 1-15 pointer 1-15, 9-5ff set in accumulator, narrow skip on any 10-146 in memory location, narrow skip on any 10-146 to one, wide skip on accumulator 2-23, 10-150 to zero, wide skip on accumulator 2-23, 10-150 three of SBR, setting 7-2 Bit to one, set 9-5ff, 10-10 skip on zero bit and set 9-5f, 9-8, 10-115f wide skip on zero bit and set 9-8, 10-121f wide skip on zero bit and set 9-8, 10-122 two of SBR, setting 7-1ff zero of fault code in AC1 9-3 narrow stack pointer 5-22f narrow stack pointer 5-22f arrow stack pointer 5-22f arrow stack pointer 5-22f sign on zero bit and set 9-8, 10-126 sin memory location, narrow stack pointer 5-22f sto zero, set 9-5ff, 10-10 skip on zero bit and set 9-8, 10-156 sto zero, set 9-5ff, 10-10f swide set 9-8, 10-122 sto of SBR, setting 7-1ff zero of fault code in AC1 9-3 narrow stack pointer 5-22f arrow stack pointer 5-22f sto zero, set 9-5ff, 10-10f state of referenced a-11, 10-103 store modified and referenced 8-11, 10-109 table of referenced a-11, 10-109 table of referenced a-11, 10-109 table of referenced a-11, 10-109 table of referenced and modified 8-1 wide count 9-8, 10-126 BKPT 2-9, 4-5, 4-8, 5-5f, 10-9 BLM 9-8, 10-10 Block add and move 9-8, 10-8f and execute, pop 2-9, 4-5, 4-8, 5-5f, 5-23, 9-10f, 10-100, 10-102 wide pop 2-9, 4-5, 4-8, 5-5f, 5-23, 9-10f, 10-100, 10-102 wide pop 2-9, 4-5, 4-8, 5-5f, 5-23, 9-10f, 10-100, 10-102 wide pop 2-9, 4-5, 4-8, 5-5f, 5-23, 9-10f, 10-100, 10-102 wide stack return 4-5 move 9-8, 10-10 move, wide stack return 4-5 move 9-8, 10-10 move, wide stack return 5-16 nar stack, accessing double word 4-4 fault return 8-13 floating-point fault return 5-16 narrow return 9-2 pop 5-23, 9-10f, 10-102 popping return 4-1, 4-5 wide return 4-1, 4-5 wide count 9-8, 10-126 bKPT 2-9, 4-5, 4-8, 5-5f, 5-23, 9-10f, 10-100, 10-102 wide pop 2-9, 4-5, 4-8, 5-5f, 5-23, 9-10f, 10-100 move, wide stack return 4-5 move 9-8, 10-10 move, wide stack ret	addressing,	
format wide 1-15 and set bit to one, skip on zero 9-5f, 9-8, 10-115f wide skip on zero 9-8, 10-156 operand 1-15 pointer 1-15, 9-5ff set in accumulator, narrow skip on any 10-98 wide skip on any 10-98 wide skip on any 10-146 in memory location, narrow skip on any 10-98 wide skip on any 10-98 wide skip on any 10-98 wide skip on any 10-146 to one, wide skip on accumulator 2-23, 10-150 to zero, wide skip on accumulator 2-23, 10-150 three of SBR, setting 7-2 Bit to one, set 9-5ff, 10-10 skip on zero bit and set 9-5f, 9-8, 10-115f wide set 9-8, 10-121f wide skip on zero bit and set 9-8, 10-156 to zero, set 9-5ff, 10-10f wide set 9-8, 10-122 two of SBR, setting 7-1ff zero of fault code in AC1 9-3 narrow stack pointer 5-22f sto one, skip on zero of skip on zero bit and set 9-8, 10-156 to zero, set 9-5ff, 10-10f side set 9-8, 10-122 two of SBR, setting 7-1ff zero of fault code in AC1 9-3 narrow stack pointer 5-22f store modified and referenced 8-11, 10-109 table of referenced and modified 8-1 wide count 9-8, 10-126 BKPT 2-9, 4-5, 4-8, 5-5f, 10-9 BLM 9-8, 10-10 Block add and move 9-8, 10-8f and execute, pop 2-9, 4-5, 4-8, 5-5f, 5-23, 9-10f, 10-100, 10-102 wide pop 2-9, 4-5, 4-8, 5-5f, 5-23, 9-10f, 10-100 move, wide 2-3, 9-8, 10-119f to transfer, starting address of 7-2 Block, C/350 return 9-2 data 4-4 fault return 8-13 floating-point fault return 5-16 narrow return 9-2 pop 5-23, 9-10f, 10-102 popping return 4-1, 4-5 wide count 9-8, 10-10 move, 9-8, 10-10 move, 9-8, 10-10 move 9-8, 10-8 and move 9-8, 10-8 and move 9-8, 10-10 move 9-8, 10-10 move, wide 2-3, 9-8, 10-119f to transfer, starting address of 7-2 Block, C/350 return 9-2 data 4-4 fault return 8-13 floating-point fault return 8-13f pushing fault return 8-13 return 4-	format C/350 9-5ff	
and set bit to one, skip on zero 9-5f, 9-8, 10-115f wide skip on zero 9-8, 10-156 operand 1-15 pointer 1-15, 9-5ff set in accumulator, narrow skip on any 10-98 wide skip on any 10-146 in memory location, narrow skip on any 10-186 wide skip on any 10-146 to one, wide skip on accumulator 2-23, 10-150 three of SBR, setting 7-2 Bit to one, set 9-5ff, 10-10 skip on zero bit and set 9-8, 10-115f wide set 9-8, 10-121f wide skip on zero bit and set 9-8, 10-156 to zero, set 9-5ff, 10-10f wide set 9-8, 10-122 two of SBR, setting 7-1ff zero of fault code in AC1 9-3 narrow stack lointer 5-22f narrow stack pointer 5-22f set ble of referenced and modified 8-1 wide count 9-8, 10-126 wide count 9-8, 10-126 slable of referenced and modified 8-1 wide count 9-8, 10-126 slable of referenced and modified 8-1 wide count 9-8, 10-126 slable of referenced and modified 8-1 wide count 9-8, 10-126 slable of referenced and modified 8-1 wide count 9-8, 10-126 slable of referenced and modified 8-1 wide count 9-8, 10-126 slad and move 9-8, 10-10 Block and and move 9-8, 10-8f and execute, pop 2-9, 4-5, 4-8, 5-5f, 5-23, 9-10f, 10-100, 10-102 wide pop 2-9, 4-5, 4-8, 5-5f, 5-23, 9-10f, 10-100, 10-102 wide pop 2-9, 4-5, 4-8, 5-5f, 5-23, 9-10f, 10-100, 10-102 wide pop 2-9, 4-5, 4-8, 5-5f, 5-23, 9-10f, 10-100, 10-102 move, wide 2-3, 9-8, 10-119f to transfer, starting address of 7-2 Block, C/350 return 9-2 data 4-4 fault return 8-13 floating-point fault return 5-16 narrow return 9-2 pop 5-23, 9-10f, 10-102 popping return 4-1, 4-5 wide count 9-8, 10-10 move, wide 2-3, 9-8, 10-119f to transfer, starting address of 7-2 Block, C/350 return 9-2 data 4-4 fault return 8-13 floating-point fault return 5-16 narrow return 9-2 pop 5-23, 9-10f, 10-102 popping return 4-1, 4-5 wide count 9-8, 10-10 move, wide 2-3, 9-8, 10-119f to transfer, starting address of 7-2 Block, C/350 return 9-2 data 4-4 fau	format wide 1-15	
skip on zero 9-5f, 9-8, 10-115f wide skip on zero 9-8, 10-156 operand 1-15 pointer 1-15, 9-5ff set in accumulator, narrow skip on any 10-98 wide skip on any 10-146 in memory location, narrow skip on any 10-186 to one, wide skip on accumulator 2-23, 10-150 to zero, wide skip on accumulator 2-23, 10-150 to one, set 9-5ff, 10-10 skip on zero bit and set 9-8, 10-12f wide set 9-8, 10-121f wide skip on zero bit and set 9-8, 10-156 to zero, set 9-5ff, 10-10f wide set 9-8, 10-122 two of SBR, setting 7-1ff zero of fault code in AC1 9-3 narrow stack pointer 5-22f narrow stack pointer 5-22f narrow stack pointer 5-22f set in accumulator, add and move 9-8, 10-18 Block add and move 9-8, 10-8f and execute, pop 2-9, 4-5, 4-8, 5-5f, 5-23, 9-10f, 10-100, 10-102 wide pop 2-9, 4-5, 4-8, 5-8f, 5-24, 9-10-f, 10-100 nove, 9-4, 5-4, 4-8, 5-8, 5-14, 5-21, 9-10f, 10-102 move 9-8, 10-10 move, wide 2-3, 9-8, 10-119f to transfer, starting address of 7-2 Block, C/350 return 9-2 data 4-4 fault return 8-13 floating-point fault return 5-16 narrow return 9-2 pop 5-23, 9-10f, 10-102 popping return 4-1, 4-5 wide return 4-1, 4-5 wide return 4-1 set out 1-4, 4-5 wide return 4-1 set out 1-4, 4-5 wide count 9-8, 10-10 Block add and move 9-8, 10-8f and execute, pop 2-9, 4-5, 4-8, 5-5f, 5-23, 9-10f, 10-100, 10-102 wide pop 2-9, 4-5, 4-8, 5-8f, 5-24, 8-5-5f, 10-9 BLM 9-8, 10-10 Block add and move 9-8, 10-16 move, wide 2-3, 9-8, 10-110 move, wide 2-3, 9-8, 10-119f to transfer, starting address of 7-2 Block, C/350 return 9-2 data 4-4 fault return 8-13 floating-point fault return 5-16 narrow return 9-2 pop 5-23, 9-10f, 10-102 popping return 4-1, 4-5 wide return 4-1, 4-5 wide return 4-1, 4-5 wide return 4-1, 4-5 wide count 9-8, 10-10	and set bit to one,	·
wide skip on zero 9-8, 10-156 operand 1-15 pointer 1-15, 9-5ff set in accumulator, narrow skip on any 10-98 wide skip on any 10-146 in memory location, narrow skip on any 10-146 to one, wide skip on accumulator 2-23, 10-150 three of SBR, setting 7-2 Bit to one, set 9-5ff, 10-10 skip on zero bit and set 9-5f, 9-8, 10-115f wide set 9-8, 10-121f wide skip on zero bit and set 9-8, 10-156 to zero, set 9-5ff, 10-10f wide set 9-8, 10-122 two of SBR, setting 7-1ff zero of fault code in AC1 9-3 narrow stack limit 5-22f narrow stack pointer 5-22f BKPT 2-9, 4-5, 4-8, 5-5f, 10-9 BLM 9-8, 10-10 Block add and move 9-8, 10-8f and execute, pop 2-9, 4-5, 4-8, 5-5f, 5-23, 9-10f, 10-100, 10-102 wide pop 2-9, 4-5, 4-8, 5-8, 5-14, 5-21, 9-10f, 10-102 move, wide y-8, 10-10 Block add and move 9-8, 10-8f and execute, pop 2-9, 4-5, 4-8, 5-5f, 5-23, 9-10f, 10-100, 10-102 wide pop 2-9, 4-5, 4-8, 5-5f, 5-23, 9-10f, 10-100, 10-102 wide pop 2-9, 4-5, 4-8, 5-5f, 5-23, 9-10f, 10-100 move, wide pop 2-9, 4-5, 4-8, 5-5f, 5-23, 9-10f, 10-100 move yide pop 2-9, 4-5, 4-8, 5-5f, 5-23, 9-10f, 10-100 move yide pop 2-9, 4-5, 4-8, 5-5f, 5-23, 9-10f, 10-100 move, wide y-8, 10-10 move y-8, 10-10 move y-8, 10-10 move y-8, 10-8f and execute, pop 2-9, 4-5, 4-8, 5-5f, 5-23, 9-10f, 10-100 move, wide 2-3, 9-8, 10-119f to transfer, starting address of 7-2 Block, C/350 return 9-2 data 4-4 fault return 8-13 floating-point fault return 5-16 narrow return 9-2 pop 5-23, 9-10f, 10-102 popping return 4-1, 4-5 wide return 4-3 protection violation return 8-13f pushing fault return 8-13 return 4-1, 4-5, 9-3 return 4-4, 9-10f, E-2	skip on zero 9-5f, 9-8, 10-115f	
operand 1-15 pointer 1-15, 9-5ff set in accumulator, narrow skip on any 10-98 wide skip on any 10-146 in memory location, narrow skip on any 10-98 wide skip on any 10-146 to one, wide skip on accumulator 2-23, 10-150 three of SBR, setting 7-2 Bit to one, set 9-5ff, 10-10 skip on zero bit and set 9-5f, 9-8, 10-115f wide set 9-8, 10-121f wide skip on zero bit and set 9-8, 10-156 to zero, set 9-5ff, 10-10f wide set 9-8, 10-122 two of SBR, setting 7-1ff zero of fault code in AC1 9-3 narrow stack limit 5-22f narrow stack pointer 5-22f BLM 9-8, 10-10 Block add and move 9-8, 10-8f and execute, pop 2-9, 4-5, 4-8, 5-5f, 5-23, 9-10f, 10-100, 10-102 wide pop 2-9, 4-5, 4-8, 5-8, 5-14, 5-21, 9-10f, 10-143 in a stack, accessing double word 4-4 in astack, accessing double word 4-4 in structions, wide stack return 4-5 move 9-8, 10-10 move, wide 2-3, 9-8, 10-119f to transfer, starting address of 7-2 Block, C/350 return 9-2 data 4-4 fault return 8-13 floating-point fault return 5-16 narrow return 9-2 pop 5-23, 9-10f, 10-102 popping return 4-1, 4-5 wide return 4-3 protection violation return 8-13f pushing fault return 8-13 return 4-1, 4-5, 9-3 return 4-4, 9-10f, E-2	wide skip on zero 9-8, 10-156	
pointer 1-15, 9-5ff set in accumulator, narrow skip on any 10-98 wide skip on any 10-146 in memory location, narrow skip on any 10-146 to one, wide skip on accumulator 2-23, 10-150 to zero, wide skip on accumulator 2-23, 10-150 three of SBR, setting 7-2 Bit to one, set 9-5ff, 10-10 skip on zero bit and set 9-5f, 9-8, 10-115f wide set 9-8, 10-121f wide skip on zero bit and set 9-8, 10-156 to zero, set 9-5ff, 10-10f wide set 9-8, 10-122 two of SBR, setting 7-1ff zero of fault code in AC1 9-3 narrow stack limit 5-22f narrow stack pointer 5-22f narrow stack	operand 1-15	
in accumulator, narrow skip on any 10-98 wide skip on any 10-146 in memory location, narrow skip on any 10-146 to one, wide skip on accumulator 2-23, 10-150 to zero, wide skip on accumulator 2-23, 10-150 skip on zero bit and set 9-5f, 9-8, 10-115f wide set 9-8, 10-121f wide skip on zero bit and set 9-8, 10-156 to zero, set 9-5ff, 10-10 wide set 9-8, 10-122 two of SBR, setting 7-1ff zero of fault code in AC1 9-3 narrow stack limit 5-22f narrow stack pointer 5-22f and execute, pop 2-9, 4-5, 4-8, 5-8, 5-14, 5-21, 9-10f, 10-102 wide pop 2-9, 4-5, 4-8, 5-8, 5-14, 5-21, 9-10f, 10-143 in a stack, accessing double word 4-4 instructions, wide stack return 4-5 move 9-8, 10-10 move, wide 2-3, 9-8, 10-119f to transfer, starting address of 7-2 Block, C/350 return 9-2 data 4-4 fault return 8-13 floating-point fault return 5-16 narrow return 9-2 pop 5-23, 9-10f, 10-102 popping return 4-1, 4-5 wide return 4-3 protection violation return 8-13f pushing fault return 8-13 return 4-1, 4-5, 9-3 return 4-4, 9-10f, E-2	pointer 1-15, 9-5ff	
in accumulator, narrow skip on any 10-98 wide skip on any 10-146 in memory location, narrow skip on any 10-98 wide skip on any 10-146 to one, wide skip on accumulator 2-23, 10-150 three of SBR, setting 7-2 Bit to one, set 9-5ff, 10-10 skip on zero bit and set 9-5f, 9-8, 10-115f wide set 9-8, 10-121f wide skip on zero bit and set 9-8, 10-156 to zero, set 9-5ff, 10-10f wide set 9-8, 10-122 two of SBR, setting 7-1ff zero of fault code in AC1 9-3 narrow stack limit 5-22f narrow stack pointer 5-22f and execute, pop 2-9, 4-5, 4-8, 5-8f, 5-23, 9-10f, 10-100, 10-102 wide pop 2-9, 4-5, 4-8, 5-8f, 5-21, 9-10f, 10-143 in a stack, accessing double word 4-4 instructions, wide stack return 4-5 move 9-8, 10-10 move, wide 2-3, 9-8, 10-119f to transfer, starting address of 7-2 Block, C/350 return 9-2 data 4-4 fault return 8-13 floating-point fault return 5-16 narrow return 9-2 pop 5-23, 9-10f, 10-102 popping return 4-1, 4-5 wide return 4-3 protection violation return 8-13f pushing fault return 8-13 return 4-4, 9-10f, E-2	set	add and move 9-8, 10-8f
narrow skip on any 10-98 wide skip on any 10-146 in memory location, narrow skip on any 10-188 wide skip on any 10-198 wide skip on any 10-198 wide skip on any 10-146 to one, wide skip on accumulator 2-23, 10-150 to zero, wide skip on accumulator 2-23, 10-150 three of SBR, setting 7-2 Bit to one, set 9-5ff, 10-10 skip on zero bit and set 9-5f, 9-8, 10-115f wide set 9-8, 10-121f wide skip on zero bit and set 9-8, 10-156 to zero, set 9-5ff, 10-10f wide set 9-8, 10-122 two of SBR, setting 7-1ff zero of fault code in AC1 9-3 narrow stack limit 5-22f narrow stack pointer 5-22f narrow stack pointer 5-22f narrow stack pointer 5-22f pop 2-9, 4-5, 4-8, 5-5f, 5-23, 9-10f, 10-100, 10-102 wide pop 2-9, 4-5, 4-8, 5-8, 5-14, 5-21, 9-10f, 10-143 in a stack, accessing double word 4-4 instructions, wide stack return 4-5 move 9-8, 10-119 to transfer, starting address of 7-2 Block, C/350 return 9-2 data 4-4 fault return 8-13 floating-point fault return 5-16 narrow return 9-2 pop 5-23, 9-10f, 10-102 popping return 4-1, 4-5 wide return 4-3 protection violation return 8-13f pushing fault return 8-13 return 4-1, 4-5, 9-3 return 4-4, 9-10f, E-2		_
wide skip on any 10-146 in memory location, narrow skip on any 10-98 wide skip on any 10-146 to one, wide skip on accumulator 2-23, 10-150 to zero, wide skip on zero bit and set 9-5f, 9-8, 10-115f wide set 9-8, 10-121f wide skip on zero bit and set 9-8, 10-156 to zero, set 9-5ff, 10-10f wide set 9-8, 10-121f wide skip on zero bit and set 9-8, 10-156 to zero, set 9-5ff, 10-10f wide set 9-8, 10-122 two of SBR, setting 7-1ff zero of fault code in AC1 9-3 narrow stack pointer 5-22f narrow stack pointer 5-22f narrow stack pointer 5-22f sind wide pop 2-9, 4-5, 4-8, 5-8, 5-14, 5-21, 9-10f, 10-143 in a stack, accessing double word 4-4 instructions, wide stack return 4-5 move 9-8, 10-119f to transfer, starting address of 7-2 Block, C/350 return 9-2 data 4-4 fault return 8-13 floating-point fault return 5-16 narrow return 9-2 pop 5-23, 9-10f, 10-102 popping return 4-1, 4-5 wide return 4-3 protection violation return 8-13f pushing fault return 8-13 return 4-4, 9-10f, E-2	narrow skip on any 10-98	
in memory location, narrow skip on any 10-98 wide skip on any 10-146 to one, wide skip on accumulator 2-23, 10-150 three of SBR, setting 7-2 Bit to one, set 9-5ff, 10-10 skip on zero bit and set 9-5f, 9-8, 10-115f wide set 9-8, 10-121f wide skip on zero bit and set 9-8, 10-156 to zero, set 9-5ff, 10-10f wide set 9-8, 10-122 two of SBR, setting 7-1ff zero of fault code in AC1 9-3 narrow stack pointer 5-22f narrow stack pointer 5-22f in a stack, accessing double word 4-4 instructions, wide stack return 4-5 move 9-8, 10-10 move, wide 2-3, 9-8, 10-119f to transfer, starting address of 7-2 Block, C/350 return 9-2 data 4-4 fault return 8-13 floating-point fault return 5-16 narrow return 9-2 pop 5-23, 9-10f, 10-102 popping return 4-1, 4-5 wide return 4-3 protection violation return 8-13f pushing fault return 8-13 return 4-1, 4-5, 9-3 return 4-4, 9-10f, E-2	wide skip on any 10-146	
narrow skip on any 10-98 wide skip on any 10-146 to one, wide skip on accumulator 2-23, 10-150 three of SBR, setting 7-2 Bit to one, set 9-5ff, 10-10 skip on zero bit and set 9-5f, 9-8, 10-115f wide set 9-8, 10-121f wide skip on zero bit and set 9-8, 10-156 to zero, set 9-5ff, 10-10f wide set 9-8, 10-122 two of SBR, setting 7-1ff zero of fault code in AC1 9-3 narrow stack pointer 5-22f narrow stack pointer 5-22f instructions, wide stack return 4-5 move 9-8, 10-10 move, wide 2-3, 9-8, 10-119f to transfer, starting address of 7-2 Block, C/350 return 9-2 data 4-4 fault return 8-13 floating-point fault return 5-16 narrow return 9-2 pop 5-23, 9-10f, 10-102 popping return 4-1, 4-5 wide return 4-3 protection violation return 8-13f pushing fault return 8-13 return 4-1, 4-5, 9-3 return 4-4, 9-10f, E-2	in memory location,	
wide skip on any 10-146 to one, wide skip on accumulator 2-23, 10-150 three of SBR, setting 7-2 Bit to one, set 9-5ff, 10-10 skip on zero bit and set 9-5f, 9-8, 10-115f wide set 9-8, 10-121f wide skip on zero bit and set 9-8, 10-156 to zero, set 9-5ff, 10-10f wide set 9-8, 10-122 two of SBR, setting 7-1ff zero of fault code in AC1 9-3 narrow stack limit 5-22f narrow stack pointer 5-22f move 9-8, 10-10 move, wide 2-3, 9-8, 10-119f to transfer, starting address of 7-2 Block, C/350 return 9-2 data 4-4 fault return 8-13 floating-point fault return 5-16 narrow return 9-2 pop 5-23, 9-10f, 10-102 popping return 4-1, 4-5 wide return 4-3 protection violation return 8-13f pushing fault return 8-13 return 4-1, 4-5, 9-3 return 4-4, 9-10f, E-2	narrow skip on any 10-98	
to one, wide skip on accumulator 2-23, 10-150 to zero, wide skip on accumulator 2-23, 10-150 three of SBR, setting 7-2 Bit to one, set 9-5ff, 10-10 skip on zero bit and set 9-5f, 9-8, 10-115f wide set 9-8, 10-121f wide skip on zero bit and set 9-8, 10-156 to zero, set 9-5ff, 10-10f wide set 9-8, 10-122 two of SBR, setting 7-1ff zero of fault code in AC1 9-3 narrow stack limit 5-22f narrow stack pointer 5-22f move, wide 2-3, 9-8, 10-119f to transfer, starting address of 7-2 Block, C/350 return 9-2 data 4-4 fault return 8-13 floating-point fault return 5-16 narrow return 9-2 pop 5-23, 9-10f, 10-102 popping return 4-1, 4-5 wide return 4-3 protection violation return 8-13f pushing fault return 8-13 return 4-1, 4-5, 9-3 return 4-4, 9-10f, E-2	wide skip on any 10-146	
to zero, wide skip on accumulator 2-23, 10-150 three of SBR, setting 7-2 Bit to one, set 9-5ff, 10-10 skip on zero bit and set 9-5f, 9-8, 10-115f wide set 9-8, 10-121f wide skip on zero bit and set 9-8, 10-156 to zero, set 9-5ff, 10-10f wide set 9-8, 10-122 two of SBR, setting 7-1ff zero of fault code in AC1 9-3 narrow stack pointer 5-22f narrow stack pointer 5-22f to transfer, starting address of 7-2 Block, C/350 return 9-2 data 4-4 fault return 8-13 floating-point fault return 5-16 narrow return 9-2 pop 5-23, 9-10f, 10-102 popping return 4-1, 4-5 wide return 4-3 protection violation return 8-13f pushing fault return 8-13 return 4-1, 4-5, 9-3 return 4-4, 9-10f, E-2	to one, wide skip on accumulator 2-23, 10-150	
three of SBR, setting 7-2 Bit to one, set 9-5ff, 10-10 skip on zero bit and set 9-5f, 9-8, 10-115f wide set 9-8, 10-121f wide skip on zero bit and set 9-8, 10-156 to zero, set 9-5ff, 10-10f wide set 9-8, 10-122 to of SBR, setting 7-1ff zero of fault code in AC1 9-3 narrow stack limit 5-22f narrow stack pointer 5-22f narrow stack pointer 5-22f setting 7-16f Block, C/350 return 9-2 data 4-4 fault return 8-13 floating-point fault return 5-16 narrow return 9-2 pop 5-23, 9-10f, 10-102 popping return 4-1, 4-5 wide return 4-3 protection violation return 8-13f pushing fault return 8-13 return 4-1, 4-5, 9-3 return 4-4, 9-10f, E-2	to zero, wide skip on accumulator 2-23, 10-150	
to one, set 9-5ff, 10-10 skip on zero bit and set 9-5f, 9-8, 10-115f wide set 9-8, 10-121f wide skip on zero bit and set 9-8, 10-156 to zero, set 9-5ff, 10-10f wide set 9-8, 10-122 to of fault code in AC1 9-3 narrow stack limit 5-22f narrow stack pointer 5-22f to one, data 4-4 fault return 8-13 floating-point fault return 5-16 narrow return 9-2 pop 5-23, 9-10f, 10-102 popping return 4-1, 4-5 wide return 4-3 protection violation return 8-13f pushing fault return 8-13 return 4-1, 4-5, 9-3 return 4-1, 4-5, 9-3 return 4-4, 9-10f, E-2	three of SBR, setting 7-2	
to one, set 9-5ff, 10-10 skip on zero bit and set 9-5f, 9-8, 10-115f wide set 9-8, 10-121f wide skip on zero bit and set 9-8, 10-156 to zero, set 9-5ff, 10-10f wide set 9-8, 10-122 two of SBR, setting 7-1ff zero of fault code in AC1 9-3 narrow stack limit 5-22f narrow stack pointer 5-22f data 4-4 fault return 8-13 floating-point fault return 5-16 narrow return 9-2 pop 5-23, 9-10f, 10-102 popping return 4-1, 4-5 wide return 4-3 protection violation return 8-13f pushing fault return 8-13 return 4-1, 4-5, 9-3 return 4-1, 4-5, 9-3 return 4-4, 9-10f, E-2	Bit	
set 9-5ff, 10-10 skip on zero bit and set 9-5f, 9-8, 10-115f wide set 9-8, 10-121f wide skip on zero bit and set 9-8, 10-156 to zero, set 9-5ff, 10-10f wide set 9-8, 10-122 two of SBR, setting 7-1ff zero of fault code in AC1 9-3 narrow stack limit 5-22f narrow stack pointer 5-22f skip on zero bit and set 9-8, 10-115f floating-point fault return 5-16 narrow return 9-2 pop 5-23, 9-10f, 10-102 popping return 4-1, 4-5 wide return 4-3 protection violation return 8-13f pushing fault return 8-13 return 4-1, 4-5, 9-3 return 4-1, 4-5, 9-3 return 4-4, 9-10f, E-2	to one,	, *
skip on zero bit and set 9-5f, 9-8, 10-115f wide set 9-8, 10-121f wide skip on zero bit and set 9-8, 10-156 to zero, set 9-5ff, 10-10f wide set 9-8, 10-122 two of SBR, setting 7-1ff zero of fault code in AC1 9-3 narrow stack limit 5-22f narrow stack pointer 5-22f floating-point fault return 5-16 narrow return 9-2 pop 5-23, 9-10f, 10-102 popping return 4-1, 4-5 wide return 4-3 protection violation return 8-13f pushing fault return 8-13 return 4-1, 4-5, 9-3 return 4-4, 9-10f, E-2	set 9-5ff, 10-10	
wide set 9-8, 10-121f wide skip on zero bit and set 9-8, 10-156 to zero, set 9-5ff, 10-10f wide set 9-8, 10-122 two of SBR, setting 7-1ff zero of fault code in AC1 9-3 narrow stack limit 5-22f narrow stack pointer 5-22f marrow return 9-2 pop 5-23, 9-10f, 10-102 popping return 4-1, 4-5 wide return 4-3 protection violation return 8-13f pushing fault return 8-13 return 4-1, 4-5, 9-3 return 4-4, 9-10f, E-2	skip on zero bit and set 9-5f, 9-8, 10-115f	
wide skip on zero bit and set 9-8, 10-156 to zero, set 9-5ff, 10-10f wide set 9-8, 10-122 two of SBR, setting 7-1ff zero of fault code in AC1 9-3 narrow stack limit 5-22f narrow stack pointer 5-22f popping return 4-1, 4-5 wide return 4-3 protection violation return 8-13f pushing fault return 8-13 return 4-1, 4-5, 9-3 return 4-4, 9-10f, E-2	wide set 9-8, 10-121f	·
to zero, set 9-5ff, 10-10f wide set 9-8, 10-122 two of SBR, setting 7-1ff zero of fault code in AC1 9-3 narrow stack limit 5-22f narrow stack pointer 5-22f return 4-1, 4-5 wide return 4-3 protection violation return 8-13f pushing fault return 8-13 return 4-1, 4-5, 9-3 return 4-4, 9-10f, E-2	wide skip on zero bit and set 9-8, 10-156	
set 9-5ff, 10-10f wide set 9-8, 10-122 two of SBR, setting 7-1ff zero of fault code in AC1 9-3 narrow stack limit 5-22f narrow stack pointer 5-22f return 4-1, 4-5 wide return 4-3 protection violation return 8-13f pushing fault return 8-13 return 4-1, 4-5, 9-3 return 4-1, 4-5, 9-3 return 4-4, 9-10f, E-2	to zero,	
wide set 9-8, 10-122 two of SBR, setting 7-1ff zero of fault code in AC1 9-3 narrow stack limit 5-22f narrow stack pointer 5-22f mide return 4-3 protection violation return 8-13f pushing fault return 8-13 return 4-1, 4-5, 9-3 return 4-4, 9-10f, E-2	set 9-5ff, 10-10f	
two of SBR, setting 7-1ff zero of fault code in AC1 9-3 narrow stack limit 5-22f narrow stack pointer 5-22f return 4-1, 4-5, 9-3 return 4-4, 9-10f, E-2	wide set 9-8, 10-122	
zero of pushing fault code in AC1 9-3 fault return 8-13 return 4-1, 4-5, 9-3 return 4-4, 9-10f, E-2		
fault code in AC1 9-3 narrow stack limit 5-22f narrow stack pointer 5-22f return 4-1, 4-5, 9-3 return 4-4, 9-10f, E-2		
narrow stack limit 5-22f return 4-1, 4-5, 9-3 narrow stack pointer 5-22f return 4-4, 9-10f, E-2	fault code in AC1 9-3	
narrow stack pointer 5-22f return 4-4, 9-10f, E-2		
100011 1,5 101, 22	narrow stack pointer 5-22f	
		stack fault return 4-8, 5-21, 5-23, 8-13

standard wide return 4-6 wide	Bytes, deleting 2-15
pop 2-9, 4-5, 4-8, 5-5f, 5-8, 5-14, 5-21, 9-10f, 10-143	inserting 2-15, 2-21f
pop context 2-9, 8-7, 8-10f, 10-129	moving 2-20
return 4-3	swapping two 2-6
Blocking an interrupt request 7-5f	
Blocks of words, between memory and device transferring	C
1-6	C/350
BMC	accumulator relative addressing 9-1
device 1-5	address space 9-4ff
maps 7-3	bit addressing, format 9-5ff
Boundary,	byte addressing 9-5f
byte 1-2	commercial faults B-2
two Kbyte address 8-3	compatibility F-2
word 1-2, 3-2	decimal and ASCII fault handler 9-3
Bracket 5-10ff	effective address 9-2, 9-5
Branch (PC relative jump) 5-2	equivalent instruction 9-8, 9-11
Branch, wide 9-10, 10-120	faults and interrupts 9-3
Breakpoint 10-9	fixed-point
Breakpoint handler 2-9, 4-5, 4-8, 5-5f	accumulator 9-1
BTO 9-5ff, 10-10	instructions 9-7f
BTZ 9-5ff, 10-10f	floating-point
Building	accumulator 9-1
a queue 6-1	fault handler 9-3
device control table 7-12	instructions 9-9
Burst multiplexor channel 1-5f, 7-1f	instructions 9-4
BUSY flag 7-3, 7-5	memory reference instruction 9-1, 9-4ff
Byte	pointers 9-3
address,	program
load effective 1-13, 2-21, 10-76, 10-166	counter 9-2
push 2-21	development 9-2ff
push effective 2-20f, 4-5f, 4-8, 10-81, 10 171	flow instructions 9-2, 9-10
addressing format 1-9f, 9-5f	registers 9-1
manipulation 2-1	return block 9-2
pointer 1-13f, 5-12, 9-5f	signed divided instructions B-2
to fault subopcode 5-19	stack 9-2
to subopcode causing fault 5-18	stack instructions 9-11
to word pointer, converting 2-21	stack-referenced instructions 9-1
pointer, skip on valid 10-116	word addressing, format 9-5
Byte,	Calculating
BCD digits per 2-17	a two's complement number 2-8, 5-14
boundary 1-2	a two's complement number, out of range 5-15
compare instructions 2-15	an effective address 2-21f
extended	the result 3-5
load 9-8, 10-38	Call
store 9-8, 10-41 indicator 1-13, 9-5f	arguments, transferring 5-10ff
load 2-20, 9-8, 10-68, 10-76, 10-165	fault code, outward E-1
move instructions 2-20	subroutine 2-9, 3-11, 4-1, 4-3, 4-5, 4-8, 5-5f, 5-8, 5-13
operand 1-13	10-65f, 10-159f
operations 2-15	Call,
packed 2-15	aborting subroutine 5-11f
skip on valid 2-22	executing subroutine 5-1
store 2-20, 2-23, 9-8, 10-85, 10-112, 10-171	illegal outward subroutine 5-9
unpacked 2-15	inward 5-8
wide	returning from 4-1, 4-5
load 2-20, 9-8, 10-136	subroutine 5-4
store 2-20, 9-8, 10-154	too many arguments segment 5-21, E-2
Bytes in memory, moving 2-15	

Calling	Checking data transfers 5-12
C/350 subroutine from a 32-bit subroutine 9-4	Checking for
sequence 5-13	stack
Calls, I/O system 7-5	overflow 5-14, 8-13
Capabilities, functional and system protection 1-1	overflow fault 4-3
Capacity, exceeding processor storage 1-2	valid data and type 5-13, 5-16
CARRY flag 2-5f, 4-6, 5-6f, 5-15, 5-18, 5-21, 8-13, 9-2	valid operations 5-13
Carry to one or zero, set 10-18	vector stack overflow 7-13
Carry,	wide stack overflow fault 4-3
complement 10-18	Checking range of the logical address 8-6
decimal 2-22, 10-31f	Checking validity bit 5-11
floating-point 3-6	CIO 7-3, 10-11
restoring 5-4	CIOI 7-3, 10-11
skip on (see ADC, ADD, AND, COM, INC, MOV, NEG, or SUB instruction)	Clearing FPSR errors 3-10, 10-43 CLM 9-8, 10-12
storing 5-7	CMP 9-8, 10-12f
Cause of page fault 8-10	CMT 9-8, 10-121 CMT 9-8, 10-13f
Causing fault,	CMV 9-8, 10-15f
byte pointer to subopcode 5-18	COB 9-8, 10-16
decimal instruction address 5-18	Code in AC1, fault 5-17ff, 9-3, E-2
instruction 5-17	Code,
Central processor identification instructions 8-11	ASCII fault 5-17
Chain,	ASCII/decimal fault 5-17
indirect addressing 1-11	decimal fault 5-17
pointer 7-13	floating-point identification 3-11
Change, sign bit 2-6	ID 3-11
Changing	loading AC1 with fault 5-19f, 9-3
a device flag 7-4	protection
C/350 subroutine 9-4	fault 8-15
current segment of execution 7-6	fault error 8-7
interrupt mask 7-6	violation fault E-1
protection mechanism 1-17	wide stack fault 5-21, E-1f
RND flag 3-4	Codes,
Channel	ASCII C-1
I/O, burst multiplexor and data 1-5f, 7-1ff	decimal or ASCII fault E-2 fault E-1f
mask, I/O 7-12 Character	violation fault E-1
compare 9-8, 10-12f	COM 2-6, 2-13f, 10-16f
compare, wide 2-23, 9-8, 10-123f	Command
depending on	I/O 7-3, 10-11
sign flag, insert 2-21	I/O immediate 7-3, 10-11
trigger, insert 2-21	Command, issuing programmed I/O 7-3
j times, insert 2-21, 10-25	Commercial faults, C/350 B-2
move 5-12, 9-8, 10-15f	Communicating with I/O controller 7-3
until true 9-8, 10-13f	Communication, interrupt 7-5
until true, wide 2-20, 9-8, 10-124f	Compare
move, wide 2-20, 9-8, 10-125f	instructions, byte 2-15
once, insert 2-21, 10-25	to limits and
scan until true, wide 2-23, 10-126f	skip 9-8, 10-12
suppress, insert 10-25	skip, wide 2-8, 9-8, 10-122
translate and	Compare two floating-point numbers (set N and Z) 3-8f,
compare 9-8, 10-18f	10-43
compare, wide 2-20, 2-23, 9-8, 10-127f	Compare,
Character, ASCII 2-16	character 9-8, 10-12f
Characters immediate, insert 2-21, 10-25 Characters,	character translate and 9-8, 10-18f
inserting or converting string of 2-20	wide character 2-23, 9-8, 10-123f wide character translate and 2-20, 2-23, 9-8, 10-127f
move 10-28	Comparing
move j 2-21	data types 2-15
Check, valid segment 8-3	divisor and dividend 3-8

DO-loop variable to constant 5-3	decimal
the gate number 5-11	numbers 2-15
wide stack pointer 4-3	to floating-point 3-3
Compatibility,	mixed number to fraction 3-3
C/350 F-2	packed decimal data 2-20
upward program 4-1, 9-1	string of characters 2-20
Complement 2-13, 10-16f	to double or single precision 3-1
CARRY flag 2-6, 10-18	to wide data 2-2
number, out of range 5-14f	Copying arguments from the source stack 5-12
Complement,	Count
add 2-4, 2-6, 10-2f	bits 9-8, 10-16
wide 10-126	bits, wide 2-13, 9-8, 10-126
wide add 10-117	Count,
Complemented source, AND with 10-6, 10-118	argument 5-7
Complementing the mantissa sign 3-7	LCALL or XCALL 4-6
Computation,	Counter wraparound, program B-1
fixed-point 1-2, 1-16, 2-1	Counter,
floating-point 1-3, 1-16, 2-15, 3-1	incrementing program 3-9
Computing	program 2-7, 2-22, 7-13
instructions, C/350 9-8f	CPU
narrow data 2-2	device 7-4ff
Concluding vector interrupt service 7-11	identification,
Condition,	C/350 load 8-11, 10-35
stack overflow or underflow 4-3	long load 8-11, 10-66
testing	narrow load 8-11, 10-93
BUSY or DONE flag and skip on 7-3	Crossing
machine 5-2	to segment zero 7-9, 7-11, 8-10, 8-13f
Conditional skip, load physical and 10-81	violation, segment 8-13f
Constant, comparing DO-loop variable to 5-3	Crossing, segment 5-6, 5-8, 5-21, 8-13f, E-2
Constants, storing device dependent 7-12	CRYTC 2-6, 10-18
Constructing double word for vector stack 7-12	CRYTO 2-6, 10-18
Contents of WFP or WSP, storing 8-13f	CRYTZ 2-6, 10-18
Contents,	CTR 9-8, 10-18f
executing accumulator 5-2	Current
modifying stack register 4-2	interrupt mask 7-12f
reserved memory 7-11	segment 1-8, 4-1, 5-1, 5-6f, 5-13, 5-20, 5-23, 7-9, 7-11,
Context block, wide pop 2-9, 8-7, 8-10f, 10-129	8-10, 8-13f, 9-2, 9-4ff
Control, transferring program 1-8, 4-2, 4-4, 7-13, 8-13	field, modifying 1-5
Controller, device 7-3ff	of execution, changing 7-6
Controlling ION flag 7-6	segment, defined 1-4
Conversion instructions, floating-point 3-3	state of processor, saving 8-7, 8-10
Conversion,	value of P 5-18f
fixed-point	CVWN 2-2, 10-19
precision 2-2	D
to floating-point 2-20	-
floating-point to fixed-point 2-20, 3-3f, B-1	DAD 2-22, 10-20
Convert	DADI 2-21, 10-20
decimal/floating-point instructions 2-20	DAPS 2-21, 10-20
FPAC data and load in memory 2-20	DAPT 2-21, 10-21
from 32-bits to 16-bits 2-2	DAPU 2-21, 10-21
instructions, decimal move and 2-20	DASI 2-21, 10-21
the four FPAC's and load in memory 2-20	Data
to 16-bit integer 10-19	and type, checking for valid 5-13, 5-16
Convert, number too large to E-2	areas, expanding 9-3
Converting	block 4-4
a decimal	channel I/O 1-5f, 7-1ff
and loading in FPAC 2-20	element 6-1
integer 2-21 byte pointer to word pointer 2-21	
data types 2-15	

element,	Decimal
dequeuing 6-1, 6-5, 10-22	add 10-20
enqueuing 6-1, 6-4, 10-39ff	and
format 2-15	alphanumeric 16-bit data, edit 9-8, 10-35f
format,	ASCII fault handler, C/350 9-3
fixed-point logical 2-12	byte operations 2-15
invalid ASCII or decimal 5-13	loading in FPAC, converting a 2-20
from the wide stack, retrieving 4-3	arithmetic
in	
	example 2-23
A (to A buffer of device) 7-3, 10-24	instructions 2-22
an area beyond the stack, overwriting 5-20	operations 2-15
B (to B buffer of device) 7-3, 10-24	arithmetic, binary coded 2-1
C (to C buffer of device) 7-3, 10-24	carry 2-22, 10-31f
the stack, storing 4-2	conversion instructions, floating-point 3-3
instructions, wide stack 4-4	data
move instructions, floating-point 3-4	fault 5-16
out	fault code E-1f
A (from A buffer of device) 7-3, 10-30	format, invalid 5-13
B (from B buffer of device) 7-3, 10-31	type, valid 2-23
C (from C buffer of device) 7-3, 10-31	data,
structure, implicit 2-15	converting packed 2-20
transfer,	invalid 1-16
checking 5-12	type 1 fault return block for 5-18f
formatted 2-20	digit and sign, representing 2-16
I/O 1-5, 7-1ff	fault
	code 5-17
requesting I/O 7-5	
type	instruction, address of 5-19
6 or 7, invalid 5-17, E-2	fixed-point subtract instructions 2-22
description, explicit 2-15	instruction address causing fault 5-18
faults 2-23	integer, converting a 2-21
indicator, explicit 2-16	move and convert instructions 2-20
type,	numbers 2-15, 2-20
explicit 2-23	or ASCII fault code 5-14, 5-16f, E-1f
invalid sign code for 5-17	results, shifting 2-22
valid ASCII or decimal 2-23	shift instructions 2-22
types 2-15ff	string, packed and unpacked 2-16f
Data,	subtract 10-31f
accessing page to read or write 8-4	to floating-point, converting 3-3
computing narrow 2-2	Decimal,
converting	fixed-point 3-3
floating-point 2-20	packed 2-15
packed decimal 2-20	unpacked 2-15
to wide 2-2	Decimal/floating-point instructions, convert and move
edit decimal and alphanumeric 16-bit 9-8, 10-35f	2-20
invalid decimal 1-16	Decrement and
loading floating-point 3-4	
	jump if nonzero 10-21
moving	skip if zero,
bytes of 2-20	C/350 9-8, 10-34
floating-point 3-4	extended 9-8, 10-36
normalizing floating-point 3-1	narrow 2-9, 9-8, 10-78, 10-167
sign magnitude 3-1	wide 2-9, 10-86, 10-174
storing floating-point 3-4	double word addressed by WSP 2-9, 4-5, 10-34
transferring I/O 7-2	Decrement word and skip, fixed-point 2-9
type	Decrementing
1 fault return block for decimal 5-18f	an intermediate exponent 3-5
2 or 3 fault return block for ASCII 5-18f	interrupt level word 7-9
of 2-16	wide stack pointer 4-3f
DCH 1-5	word in stack 2-21
DCT 7-12	Defer (indirect) violation fault code E-1
DDTK 2-21, 10-21	
· · · · · · · · · · · · · · · · · · ·	

Defining	driver 7-2ff
limits of narrow stack 9-2	flags 7-3ff
stack location 4-2	independent operations 7-3
Deleting bytes 2-15	instruction, CPU 7-6
Demand paging,	interrupt
defined 8-10	routine 7-12f
mapping and F-1	system 7-5
DEND 2-21, 5-6, 10-22	management 1-5, 7-1
Dependent information, interrupt routine 7-12	map 7-2f, 10-137
	number, interrupting 7-12f
Depending on S, add to P 10-20	state 7-5
sign flag, insert character 2-21	Device,
T, add to P 10-21	access 7-1
,	BMC 1-5
trigger, insert character 2-21	identifying unique 7-4
Depth fault, page table 8-6	idle 7-5
DEQUE 6-5, 10-22	internal 7-4
Dequeue 10 22	sending physical address to 7-2
a queue data element 10-22	starting a 7-5
data element 6-5	vector on interrupting 10-172
Dequeuing data element 6-1	
DERR 10-23	Devices, controlling 7-3
Descriptor,	
original 5-18f	device flags for general 7-5
queue 6-2	supporting 7-1
Destination	DHXL double hex shift left 2-22, 10-23
indicator for	DHXR double hex shift right 2-22, 10-23f
STIX 5-18f	DI, add to 10-20
WSTI 5-18f	DIA 7-3, 10-24
indicator, add integer to 2-21	DIB 7-3, 10-24
segment 5-6f, 5-10ff	DIC 7-3, 10-24
segment,	DICI 2-21, 10-25
access privileges 5-12	Dictionary, instruction 10-1
accessing 5-9f	Digit
defined 1-9	and sign, representing decimal 2-16
stack overflow 5-12	with overpunch, move 2-21, 10-29
Detected error 10-23	Digit,
Detecting	invalid 5-17, E-2
a fault 2-10, 3-11, 5-1, 8-13f	significant 2-17
an error 5-13, 8-12	Digits, BCD 2-16f
an I/O interrupt request 5-1	DIMC 2-21, 10-25
an overflow fault 2-10	DINC 2-21, 10-25
nonprivileged fault 5-13	DINS 2-21, 10-25
power failure 7-5	DINT 2-21, 10-25
privileged fault 5-13	Direct address 1-15, 8-6
proper power voltage ranges 7-5	Direction of I/O transfer 5-8, 7-2
protection violation 7-2	Disable,
wide stack fault 5-20	fixed-point trap 2-9, 10-60
Detection,	floating-point trap 3-10, 10-60
enabling floating-point fault 3-11	Disabling
enabling vector stack underflow or overflow 7-9	data channel and BMC maps 7-3
Determining address of fault instruction 5-16ff	I/O interrupt recognition 7-5, 7-12
Development, supporting C/350 program 9-2	stack fault 4-7, 5-20, 5-22
Device	Disk resident page 8-1, 8-4
address field 7-4	Disk,
address translation 7-1, 7-6	loading referenced page from 8-10
and accumulator, transfers between 7-1	transferring page to 8-10
and memory, transfers between 7-1	Dispatch 5-2, 9-10, 10-32f, 10-69f
control table 7-12f	Displacement 1-11
controller 7-4f	Displacement, XVCT 7-11
dependent constants or variables, storing 7-12	DIV 2-5, 10-26
dependent constants of fariables, storing , 12	,

Divide	DONE flag 7-3, 7-5
by zero flag, mantissa 3-11	Double
double (FPAC by FPAC) 3-8, 10-44	hex shift
double (FPAC by memory) 3-8, 9-9, 10-44, 10-71,	left, DHXL 2-22, 10-23
10-161	right, DHXR 2-22, 10-23f
instruction, floating-point 3-11	logical shift 2-14, 10-27
memory word,	precision 3-1f, 3-6f
narrow 2-5, 10-78, 10-167	to single, floating-point rounding 3-3, 10-53f
wide 2-5, 10-85f, 10-173	word addressed by WSP,
single (FPAC by FPAC) 3-8, 10-45	decrement 2-9, 4-5, 10-34
single (FPAC by memory) 3-8, 9-9, 10-44, 10-72,	increment 2-9, 4-5, 10-64
10-162	load accumulator with 4-5
Divide,	store accumulator in 2-3, 4-5
narrow 10-93	word
narrow sign extend 2-5	block in a stack, accessing 4-4
sign extend and 2-5, 10-27	for vector stack, constructing 7-12
signed 2-5, 10-26	in a stack, accessing 4-4
skip on no	onto vector stack, pushing 7-12
overflow and no zero (OVF and DVZ = 0) 3-9,	operand 1-2, 1-13
10-58	word,
underflow and no zero (UNF and DVZ = 0) 3-9,	load accumulator with 10-67
10-58	pushing a 4-3
no zero (DVZ = 0) 3-9, $10-57$	Double,
unsigned 2-5, 10-26	add (FPAC to FPAC) 3-6, 10-42
wide 2-5, 10-128	add (memory to FPAC) 3-6, 9-9, 10-42, 10-71, 10-161
wide signed 2-5, 10-129	divide (FPAC by FPAC) 3-8, 10-44
Divided instructions, C/350 signed B-2	divide (FPAC by memory) 3-8, 9-9, 10-44, 10-71,
Dividend 3-8	10-161
Division	load floating-point 9-9, 10-47, 10-72, 10-162
by zero, attempting 2-8, 5-14f	multiply (FPAC by FPAC) 3-7, 10-49f
instructions,	multiply (FPAC by memory) 9-9, 10-49, 10-73f, 10-77,
fixed-point 2-5	10-163
floating-point 3-8	store floating-point 9-9, 10-59, 10-75, 10-164
Division,	subtract (FPAC from FPAC) 3-7, 10-55, 10-58
aborting floating-point 3-11	subtract (memory from FPAC) 3-7, 9-9, 10-56f, 10-74,
floating-point 3-8	10-163
Divisor	Double-word stack access instructions 4-5
and dividend, comparing 3-8	Driver, device 7-2, 7-5
equals zero 3-11	DSB 2-22, 10-31f
exponent 3-8	DSPA 9-10, 10-32f
for zero, testing the 3-8	DSSO 2-21, 10-33
Divisor, defined 3-8	DSSZ 2-21, 10-33
DIVS 2-5, 10-26	DSTK 2-21, 10-33
DIVX 2-5, 10-27	DSTO 2-21, 10-34
DLSH 2-14, 10-27	DSTZ 2-21, 10-34
DMVA 2-21, 5-17, 10-27, E-2	DSZ 9-8, 10-34
DMVC 2-21, 5-17, 10-28	DSZTS 2-9, 4-5, 10-34
DMVF 2-21, 5-17, 10-28, E-2	DVZ flag 3-8f, 3-11, 5-15
DMVN 2-21, 5-17, 10-29, E-2	2 · 2 mg 5 oi, 5 m, 5 m
DMVO 2-21, 5-17, 10-29, E-2	E
DMVS 2-21, 5-17, 10-30, E-2	EGY VD 6 44 44 44
DNDF 2-21, 10-30	ECLID 8-11, 10-35
DO until greater	ECLIPSE C/350 code 9-3f
than instructions 5-3	EDIT 5-16f, 9-8, 10-35f, E-2
than,	Edit decimal and alphanumeric 16-bit data 9-8, 10-35f
narrow 10-78, 10-167	EDIT instruction 2-21, 9-3
wide 10-86, 10-173f	Edit subprogram 2-21, 5-4
DOA 7-3, 10-30	Edit,
DOB 7-3, 10-30	end 10-22
DOC 7-3, 10-31	wide 4-8, 5-5, 9-8, 10-129ff

EDSZ 9-8, 10-36	Entry,
Effective	addressing an 7-12
address 1-10f, 1-16, 2-20f, 5-2, 5-4f, 5-10, 7-6, 7-11,	page table 8-1ff
8-2	vector table 7-12
address,	Equivalent instruction 9-7ff
C/350 9-2, 9-5	Error
calculating an 2-21f	code, protection fault 8-7
extended load 9-10, 10-39	flag, setting an 3-6
load 2-12ff, 2-21f, 8-3, 9-10, 10-70, 10-76, 10-165	Error,
push 2-21	detecting an 5-13, 8-12, 10-23
addressing 1-13	identifying floating-point 3-10f
addressing, C/350 9-5f	Errors, clearing FPSR 3-10, 10-43
byte address,	ESTA 9-8, 10-41
load 1-13, 2-21, 10-76, 10-166	ESTB 9-8, 10-41
	Example of one- or two-level page table translation 8-8f
push 2-20f, 4-5f, 4-8, 10-81, 10-171	Example,
logical address 8-6	decimal arithmetic 2-23
relative address 1-11	wide stack operation 5-7f
Eight segment base registers, specifying 8-6	Exceeding processor storage capacity 1-2
EISZ 9-8, 10-37	Excess 64
EJMP 9-10, 10-37	
EJSR 9-10, 10-37	notation, maintaining 3-7f
ELDA 9-8, 10-38	representation 3-2
ELDB 9-8, 10-38	Exchange accumulators 2-3, 10-160
ELEF 9-10, 10-39	Exchange, wide 2-3, 10-158
Element,	Exclusive OR 2 125 10 170
data 6-1	OR 2-12f, 10-170
dequeue data 10-22	immediate 2-13, 10-170
dequeuing or enqueuing a data 6-4f	immediate, wide 2-13, 10-159
index to a gate 5-11	OR, wide 2-13, 10-159
queue 5-4	Execute 10-160f
Empty queue 6-3	access 1-9
Enable mask, trap 3-9ff, 5-15	flag 8-5
Enable,	violation 8-13f
fixed-point trap 2-9, 10-60	accumulator instruction 5-2
floating-point trap 3-10, 10-60	instruction, accessing page to 8-4
Enabling	violation fault code E-1
fixed-point fault recognition 2-10, 5-14	Execute,
floating-point fault	pop block and 2-9, 4-5, 4-8, 5-5f, 5-23, 9-10f, 10-100,
detection 3-11	10-102
recognition 3-9ff, 5-15	wide pop block and 2-9, 4-5, 4-8, 5-8, 5-14, 5-21,
I/O	9-10f, 10-143
instruction execution 7-1ff	Executing
interrupt recognition 7-5, 7-12	16-bit instructions 9-1
narrow stack underflow 9-2	accumulator contents 5-2
OVK mask 4-8	C/350 memory reference instruction 9-4ff
stack fault recognition 5-20	floating-point instructions 3-4
TE flag 4-8	interrupted program 5-14
vector stack underflow or overflow detection 7-9	I/O instruction 7-1ff
Encountering an invalid digit 5-17	jump instruction 5-1
End	LEF instruction 7-1ff
edit 2-21, 10-22	return instruction 5-1
float 2-21, 10-30	skip instruction 5-1
ENQH 6-5, 10-39f	subroutine 5-1, 5-12
ENQT 6-5, 10-391 ENQT 6-5, 10-40	XCT instruction 5-1
Enqueue towards the head or tail 10-39f	Execution,
Enqueuing data element 6-1, 6-4	aborting instruction 2-23, 8-2
	changing current segment of 7-6
Entry point to a segment 5-9f	enabling I/O instruction 7-1ff
	normal program 5-14f
	Exiting subroutine 4-3
	2

Expanding	Fault
beyond 64 Kbytes 9-3	address parameter, vector stack 7-9
data areas 9-3	code,
ECLIPSE C/350 code 9-3f	access violation E-1
the instruction set 4-5	ASCII or decimal 5-14, 5-16f, E-2
to 32-bit fixed-point arithmetic 9-3f	decimal or ASCII 5-14, 5-16f, E-1f
Explanation of fault code E-1	explanation of E-1
Explicit data types 2-15ff, 2-23	invalid microinterrupt return block E-
Exponent 3-2	inward return E-1
into FPAC, loading 3-3	loading AC1 with 5-19f, 9-3
overflow 3-6, 3-10	nonprivileged 5-14, E-1f
	outward call E-1
underflow 3-6, 3-10	privileged E-1
Exponent,	• •
decrementing an intermediate 3-5	privileged instruction violation E-1
dividend 3-8	protection 8-7, 8-15
divisor 3-8	read violation E-1
intermediate floating-point 3-7	returned in AC1 5-17ff, 9-3, E-2
load 10-45	validity bit protection E-1
producing an intermediate 3-8	wide stack 5-21, E-1f
Exponents 3-7	write violation E-1
Extend	detection, enabling floating-point 3-11
16-bits to 32-bits,	flags,
sign 2-2, 10-108	fixed-point 2-10
zero 2-2, 10-176	floating-point 3-10, 5-15
and divide, sign 2-5, 10-27	handler,
divide, narrow sign 2-5	C/350 decimal and ASCII 9-3
multiply, narrow sign 2-5	C/350 floating-point 9-3
Extended	decimal and ASCII 5-17
add	first instruction of 5-14f, 5-20, 5-23
immediate 2-4, 10-5	first instruction of interrupt 7-13
	first instruction of vector stack 7-13
immediate, narrow 2-4, 10-91	
decrement and skip if zero 9-8, 10-36	fixed-point 2-10
displacement 1-11	floating-point 3-9, 5-16
increment and skip if zero 9-8, 10-37	jumping to 5-13ff, 8-13
jump 9-10, 10-37	jumping to narrow stack 5-23
jump to subroutine 9-10, 10-37	last instruction of 5-14, 5-21, 5-23
load	narrow stack 5-22f
accumulator 9-8, 10-38	page 5-13, 8-10f
byte 9-8, 10-38	protection violation 8-13f
effective address 9-10, 10-39	returning from 3-11, 4-1
operation 4-5, 4-8, 9-10f, 10-169f	saving pointer to wide stack 7-9
operation, wide 5-5, 9-10f, 10-102, 10-158	vector stack 7-13
store accumulator 9-8, 10-41	wide stack 5-20f, 5-23
store byte 9-8, 10-41	handling, first instruction after 5-15f
Extended,	instruction,
load integer 9-9, 10-69	address of 5-14ff, 5-20f, 5-23
store integer 9-9, 10-113	address of decimal 5-19
wide load integer 9-9, 10-136f	address of WEDIT 5-18
wide store integer 9-9, 10-155	determining address of 5-16ff
External device 7-4	setting AC0 with address of 8-13
External device 7-4	mask,
F	·
•	setting fixed-point overflow 5-14
FAB 3-3, 10-42	setting floating-point 5-15
Facilities, maintaining I/O 7-5	TE 5-15
FAD 3-6, 10-42	operations, stack 5-20f
Failure, detecting power 7-5	pointer,
FAMD 9-9, 10-42	initializing wide stack 7-9
FAMS 9-9, 10-42f	privileged and nonprivileged 5-13
FAS 3-6, 10-43	
1110 5 0, 10 15	

recognition,	setting AC1 with value identifying 8-13
enabling fixed-point 2-10, 5-14	stack 4-1, 5-12, 5-14
enabling floating-point 3-9ff, 5-15	types of 1-16
enabling stack 5-20	wide
return block,	floating-point 5-15f
fixed-point 5-15	stack 5-20
floating-point 5-16	words required beyond WSL for stack 4-8
pushing 5-14, 5-19, 8-13	and interrupts, C/350 9-3
	Faults,
stack 4-8, 5-21, 5-23, 8-13	C/350 commercial B-2
return blocks,	data type 2-23
types of narrow 5-19	floating-point 3-9, 5-15
types of wide 5-18	handling 5-13
service mask 1-2f	priorities on 8-12f
subopcode, byte pointer to 5-19	priority of handling 5-13
Fault,	protection violation 8-12ff
aborting nonprivileged 5-14, 8-13f	
ASCII data 5-14, 5-16	type of stack 5-20
byte pointer to subopcode causing 5-18	wide stack 4-7
cause of page 8-10	FCLE 3-10, 10-43
checking for stack overflow 4-3	FCMP 3-8f, 10-43
decimal	FDD 3-8, 10-44
data 5-16f	FDMD 9-9, 10-44
instruction address causing 5-18	FDMS 9-9, 10-44
or ASCII E-2	FDS 3-8, 10-45
detecting	FEXP 3-3, 10-45
a 2-10, 3-11, 5-1, 8-13f	FFAS 3-3, 3-11, 10-45
an overflow 2-10	FFMD 3-11, 9-9, 10-46
floating-point 5-15	FHLV 3-8, 10-46
nonprivileged or privileged 5-13	Field in context block, status 8-10
wide stack 5-20	Field,
disabling stack 4-7, 5-20, 5-22	data type and size 2-16
fixed-point overflow 1-2, 2-8, 5-13ff	device address 7-4
floating-point 3-9, 5-13f	immediate 8-7
ignoring floating-point overflow 5-15	Final
	intermediate mantissa 3-7
initiating	interrupt processing 7-11
a protection 5-8	Finishing an I/O operation 7-5
fixed-point overflow 5-14	FINT 3-3f, 10-46f
floating-point 5-15	First
instruction causing 5-17	64 Kbytes of segment 9-2
microinterrupt or 5-21	entry in page table, address of 8-2ff
narrow stack 5-20f	instruction after fault handling 5-15f
nonprivileged 1-16f, 5-13	instruction of
nonresident page 5-13	fault handler 5-14f, 5-20, 5-23
page 8-4, 8-7, 8-10	
table depth 8-6	interrupt fault handler 7-13
table validity protection 8-7	interrupt handler 7-7, 9-3
privileged 1-16f	subroutine, address 5-10, 5-12
privileged access 5-12	vector stack fault handler 7-13
protection 1-8f, 2-22, 5-11, 5-13, 8-6f	word for I/O transfer, address of 7-2
return address to original 8-13	word of interrupt handler 7-7
segment validity protection 8-2	Fix
servicing 1-2, 2-10	from floating-point accumulator, wide 3-3f, 3-11, 9-9
a floating-point 3-9	10-131
floating-point 3-9, 3-11, 5-15	to accumulator 10-45
nonprivileged 5-14	to memory 9-9, 10-46
original 8-13	Fixed-point
overflow 2-10	accumulator 1-2
page 8-4, 8-10	accumulator,
protection violation 8-13f	C/350 9-1
stack 5-20, 5-22	wide float from 3-3f, 9-9, 10-131
SIAUN 3-20, 3-42	, ,

addition instructions 2-4, 2-22	error status 3-10
arithmetic instructions 2-4	execute access 8-5
arithmetic, expanding to 32-bit 9-3f	exponent overflow or underflow 3-10
byte movement instructions 2-20	indirect 1-15
computation 1-2, 1-16, 2-1	initializing
computation fault 5-13	BUSY 7-3
conversion, floating-point to 2-20, B-1	DONE 7-3
data	OVR 5-15, 7-13
format 2-1	insert character depending on sign 2-21
movement instructions 2-3	instructions to manipulate modified or referenced 8-11
precision 2-2	interrupt on 7-4ff
decimal 3-3	I/O validity 1-17, 7-2, 8-3
decrement word and skip 2-9	ION 7-4ff
division instructions 2-5	IRES 2-10f, 5-15, 5-20, 7-6, 7-13, 8-13, 9-3
fault	IXCT 2-10f
recognition, enabling 2-10, 5-14	LEF mode 8-3
return block 5-15	manipulating
increment word and skip 2-9	a device 7-5
instructions, C/350 9-7f	
logical	an interrupt on 7-5 modified 8-11
data format 2-12	· -
instructions 2-12	mantissa
	divide by zero 3-11
operations 2-12	overflow 3-11
skip instructions 2-14	memory resident 8-4
move instructions 2-3	modified 8-11
multiplication instructions 2-5	MOF 3-11, 5-15
overflow 2-6	N (negative flag) 3-8, 3-11
condition 2-10	overflow 2-8, 2-10, 5-20, 7-13
fault 1-2, 2-8, 5-14f	OVF 3-6, 3-10, 5-15
flag, setting 5-14	OVR 2-8, 2-10, 5-14, 5-20, 7-13, 8-13
mask, OVK 2-10, 9-10f	power fail 7-4f
precision conversion 2-2	read access 8-5
registers 1-2	referenced 8-11
skip on condition 2-7f	RND or round 3-4f, 3-11
subtraction	segment validity 8-2
instructions 2-4	setting
instructions, decimal 2-22	an error 3-6
to floating-point conversion 3-3	BUSY 7-5
trap	DONE 7-5
disable 2-9, 10-60	DVZ 3-8
enable 2-9, 10-60	fixed-point overflow 5-14
two's complement 2-2	ION 7-6
Flag	modified 8-11
handling, optional device 7-3	N 3-8
is zero, opcode pointer if sign 2-21	overflow 2-8
reset, skip on OVR 2-8, 10-110	OVR 8-13
to one or zero, set sign 2-21, 10-33	
Flag,	referenced 8-11
@ 9-5f	RND 3-11
\circ	Z (true zero flag) 3-8, 3-11
ANY 3-10f, 5-15	TE 4-8
BUSY 7-3, 7-5	testing
CARRY 2-5f, 4-6, 5-6f, 5-15, 5-18, 5-21, 8-13, 9-2	a device 7-3f
changing	BUSY 7-3
a device 7-4	DONE 7-3
RND 3-4	DVZ 3-9
controlling ION 7-6	translation level 8-3
device 7-3ff	UNF 3-10, 5-15
DONE 7-3, 7-5	valid page access 8-4
DVZ 3-8f, 3-11, 5-15	write access 8-5
enabling TE 4-8	Z (true zero flag) 3-11

Flags,	instructions,
device 7-3ff	C/350 9-9
floating-point fault 3-10, 5-15	executing 3-4
for general I/O devices 7-5	move instructions 3-3
for skip instruction 7-5	multiplication 3-7
setting floating-point fault 5-15	multiply double or single precision 3-7
testing status 3-8	normalize 3-3
FLAS 3-3, 10-47	number, scaling 3-3, 10-54
FLDD 9-9, 10-47	numbers, compare two (set N and Z) 3-9
FLDS 9-9, 10-47	numerical algorithms B-2
FLMD 9-9, 10-48	overflow faults 5-15
Float from	pop, wide 3-4, 4-5, 4-8, 10-131f
accumulator 10-47	program counter 3-11
accumulator, wide 3-3f, 9-9, 10-131	push, wide 3-4, 4-5, 4-8, 10-132f
memory 9-9, 10-48	result, storing the 3-6
Float,	revision 3-11
	rounding double to single 3-3, 10-53f
end 2-21, 10-30 move 10-28	single,
	load 9-9, 10-47, 10-72, 10-162
move j 2-21	store 9-9, 10-59, 10-75, 10-164
Floating-point	
accumulator,	skip on condition instructions 3-9
C/350 9-1	state,
wide fix from 3-3f, 3-11, 9-9, 10-131	pop 3-10, 9-9, 10-51f
addition instructions 3-6	push 3-10, 9-9, 10-52f
carry 3-6	status 3-9
computation 1-3, 1-16, 2-15, 3-1	status and accumulators,
computation fault 5-13	pop 10-51f
conversion 3-4	push 10-52f
conversion, fixed-point to 2-20, 3-3f	status
data,	instructions 3-10
converting 2-20	register 1-3, 3-4, 3-6, 3-9f, 9-9, 10-59, 10-74
loading 3-4	status,
moving 3-4	load 10-73
normalizing 3-1	store 5-16
storing 3-4	subtraction 3-7
decimal conversion instructions 3-3	to decimal, converting 3-3
divide instruction 3-11	to fixed-point conversion 2-20, B-1
division 3-8	to fixed-point, conversion 3-3
division, aborting 3-11	trap 3-10, 10-60, B-2
double,	underflow faults 5-15
load 9-9, 10-47, 10-72, 10-162	word, reading high 3-3
store 9-9, 10-59, 10-75, 10-164	Floating-point,
error, identifying 3-10f	conversion fixed-point to 3-3
exponent, intermediate 3-7	integerize 3-3, 10-46f
fault 3-9, 5-16	move 10-50
detection, enabling 3-11	scale 3-3
flags 3-10	FLST 9-9, 10-48
flags, setting 5-15	FMD 3-7, 10-49f
handler 3-9, 5-16	FMMD 9-9, 10-49
handler, C/350 9-3	FMMS 9-9, 10-49
mask, setting 5-15	FMOV 3-4, 10-50
recognition, enabling 3-9ff, 5-15	FMS 3-7, 10-50
return block 5-15f	FNEG 3-3, 10-50
	FNOM 3-1, 3-3, 10-50
fault,	
initiating 5-15	FNS 5-3, 10-51
servicing 3-9, 3-11, 5-15	Format,
format 3-2	ASCII and decimal data 2-15ff
halve (FPAC/2) 3-8, 10-46	byte pointer 1-14
identification code 3-11	data 2-15
instruction, address of 3-11	double precision 3-2

fixed-point	FSS 3-7, 10-58
data 2-1	FSST 5-16, 9-9, 10-59
logical 2-12	FSTD 9-9, 10-59
floating-point 3-2	FSTS 9-9, 10-59
map register assignments and 7-2	FTD 3-10, 5-15, 10-60
normalized F-3	FTE 3-10, 5-15, 10-60
processor status register 2-10	FTE mask 3-11
program counter 1-5	Functional capabilities 1-1
single precision 3-2	FXTD 2-9, 5-14, 10-60
Formatted data, transferring 2-20	FXTE 2-9f, 5-14, 10-60
Forming	·
bit pointer 9-5f	G
byte address 9-5f	
physical address 7-2	Gate
word pointer 9-5f	array 5-10f
Forward	element, index to a 5-11
link 6-2	fault code, illegal E-1
search queue and skip,	number 5-10f
narrow 5-3, 6-5, 10-95f	Gate,
wide 5-3f, 6-5, 10-133f	indexed 5-11
Four	valid 5-11
FPAC's and load in memory, convert the 2-20	General I/O
wide stack registers, initializing 7-9, 7-11	devices, device flags for 7-5
FPAC data and load in memory, convert 2-20	instructions 7-3f
FPAC,	Glossary F-1
converting a decimal and loading in 2-20	Guard digit 3-4f
loading exponent into 3-3	
FPOP 9-9, 10-51f	Н
FPSH 9-9, 10-52f	Halve,
FPSR 3-9f	floating-point (FPAC/2) 3-8, 10-46
	narrow (AC/2) 2-5, 10-40
FPSR errors, clearing 3-10, 10-43	wide (AC/2) 2-5, 10-01
Frame pointer,	Handler,
initializing wide 4-3, 5-11	breakpoint 2-9, 4-5, 4-8, 5-5f
loading wide 5-12f narrow 9-2	C/350
	decimal and ASCII fault 9-3
saving wide 7-9	floating-point fault 9-3
storing wide 5-7, 5-11, 5-13, 8-10	fault 2-10, 5-17, 8-7
wide 4-2f, 5-6ff, 7-9	first
FRDS 3-3, 10-53f	instruction of fault 5-14f, 5-20, 5-23
Frequency of references to pages 8-11	
FRH 3-3, 10-54	instruction of interrupt 7-7, 9-3 instruction of interrupt fault 7-13
FSA 5-3, 10-54	instruction of interrupt fault 7-13
FSCAL 3-3f, 3-11, 10-54	word of interrupt 7-7
FSD 3-7, 10-55, 10-58	
FSEQ 3-8f, 10-55	floating-point fault 3-9, 5-16
FSGE 3-9, 10-55	immediate interrupt 7-7
FSGT 3-8f, 10-55	interrupt 7-5f, 10-172
FSLE 3-9, 10-55	jumping to
FSLT 3-8f, 10-56	fault 5-13ff, 8-13
FSMD 9-9, 10-56	narrow stack fault 5-23
FSMS 9-9, 10-56	last instruction of
FSND 3-9, 10-57	fault 5-14, 5-21, 5-23
FSNE 3-9, 10-57	vectored interrupt 7-7
FSNER 3-9, 10-57	narrow stack fault 5-22f
FSNM 3-9, 10-57	page fault 5-13, 8-10f
FSNO 3-9, 10-57	protection violation fault 8-13f
FSNOD 3-9, 10-58	returning from
FSNU 3-9, 10-58	breakpoint 4-5
FSNUD 3-9, 10-58	fault 3-11, 4-1
FSNUO 3-9, 10-58	saving pointer to wide stack fault 7-9

subroutine 2-10	Immediate,
vector stack fault 7-13	add 2-4, 10-5f
vectored interrupt 7-7, 10-172	AND 2-13, 10-8
wide stack fault 5-20f	command I/O 7-3, 10-11
Handling	exclusive OR 2-13, 10-170
faults, priority of 5-13	extended add 2-4, 10-5
I/O interrupts, priority of 5-14	inclusive OR 2-13, 10-64
Handling, first instruction after fault 5-16	insert characters 2-21, 10-25
Head, enqueue towards the 10-39f	narrow
Hex	add 2-4, 10-77, 10-92, 10-166
digit 3-4	extended add 2-4, 10-91
shift left,	load 2-3, 10-96
DHXL double 2-22, 10-23	subtract 10-79, 10-98, 10-168
	subtract 2-4, 10-107
HXL single 2-22, 10-61	wide
shift right,	add 2-4, 10-85, 10-117f, 10-173
DHXR double 2-22, 10-23f	add with narrow 10-142
HXR single 2-22, 10-61	add with marrow 10 142 add with wide 10-117
Hierarchical	AND 2-13, 10-118
interrupt system 7-5	arithmetic shift with narrow 10-119
protection mechanism 1-17	
High	exclusive OR 2-13, 10-159
floating-point word, reading 3-3	inclusive OR 2-12f, 10-135
speed device and memory, transfers between 7-1	load with wide 10-135
word, read 10-54	logical shift 2-14, 10-139
HLV 2-5, 10-61	logical shift left 2-14
Honoring interrupts 7-13	logical shift with narrow 10-138
HXL single hex shift left 2-22, 10-61	skip if accumulator greater than 2-8, 10-150
HXR single hex shift right 2-22, 10-61	skip if accumulator less than or equal to 2-8, 10-15
	skip if accumulator not equal to 2-8, 10-152
1	skip if equal to 2-8
TD 1 2 11	subtract 10-88, 10-148, 10-175
ID code 3-11	unsigned skip if accumulator greater than 2-8,
Identical segments, testing for 5-12	10-156
Identification	unsigned skip if accumulator less than or equal to
code, floating-point 3-11	2-8, 10-157
instructions 8-11	Implicit data structure 2-15
Identification,	Impure zero F-2
C/350 load CPU 8-11, 10-35	INC 2-4, 2-6, 2-8f, 10-62f
long load CPU 8-11, 10-66	Inclusive OR 10-63
narrow load CPU 8-11, 10-93	immediate 2-13, 10-64
Identifying	immediate, wide 2-12f, 10-135
fault, setting AC1 with value 8-13	Inclusive OR, wide 2-13, 10-135
floating-point error 3-10f	Increasing accuracy of result 3-4
unique device 7-4	Increment
Idle device 7-5	and skip 10-62f
Ignoring	
an interrupt request 7-5	and skip if zero,
overflow faults 5-14f	C/350 9-8, 10-64
page protocol access 8-6f	extended 9-8, 10-37
Illegal	narrow 2-9, 9-8, 10-79, 10-168
gate fault code E-1	wide 2-9, 10-87, 10-174
inward return 5-9	double word addressed by WSP 2-9, 4-5, 10-64
I/O operation 8-3	with optional skip 2-4, 2-6, 2-8f
outward subroutine call 5-9	word and skip, fixed-point 2-9
	Increment, wide (no skip) 2-4, 10-135
Immediate	Incrementing
field 8-7	a DO-loop variable 5-3
interrupt handler 7-7	interrupt level word 7-9
	program counter 3-9, 5-1
	wide stack pointer 4-3f
	Independent operations, device 7-3

Index	Inserting
field 1-10	additional bytes 2-21
to a gate element 5-11	bytes 2-15
Indexed gate 5-11	string of characters 2-20
Indicator for	Instruction
STIX, destination 5-18f	address causing fault, decimal 5-18
WSTI, destination 5-18f	after fault handling, first 5-15f
Indicator,	causing fault 5-17
byte 1-13, 9-5f	dictionary 10-1
destination 2-21	execution,
explicit data type 2-16	aborting 2-23, 8-2
original source 5-18f	enabling I/O 7-1ff
source 2-21	of fault handler,
Indirect 8-2	first 5-14f, 5-20, 5-23
addressing 1-8, 1-10f, 1-15, 8-6, 9-5f	last 5-14, 5-21, 5-23
addressing violation 8-13f	of interrupt
bit 9-5f	fault handler, first 7-13
field 1-11	handler, first 7-7, 9-3
flag 1-15	handler, last 7-7
logical address 8-6	of subroutine,
pointer 1-11, 5-10, 5-13ff, 5-17, 5-20, 5-23, 8-13f, 9-7,	address first 5-10, 5-12
9-9ff	last 5-13
relative address 1-11	of vector stack fault handler, first 7-13
Indirect, jumping 3-9, 7-7, 8-10	of vectored interrupt handler, last 7-7
Indirection, levels of 5-13, 8-13f	opcodes,
Information, interrupt routine dependent 7-12	I/O 7-4
Initializing	wide B-1
burst multiplexor channel transfer 7-2	operation, I/O 7-4
BUSY flag 7-3	prefix, X or L 1-11 set F-2
carry flag instructions 2-6	
data channel transfer 7-2	set, expanding the 4-5
device map 7-2	summary A-1ff
DONE flag 7-3	violation fault code, privileged E-1
four wide stack registers 7-9, 7-11	violation, I/O 8-13f Instruction,
IRES flag 7-13 narrow stack 5-22	aborting memory reference 8-2
	accessing page to execute 8-4
OVK mask 5-15, 7-13 OVR flag 5-15, 7-13	accumulator
wide frame pointer 4-3, 5-11	jump 5-2
wide stack 4-3	skip 5-2
base 4-2, 8-10, 8-13f	address of
fault pointer 7-9	decimal fault 5-19
limit 8-10, 8-13f	fault 5-15, 5-17, 5-21, 5-23
pointer 4-3, 8-10, 8-13f	floating-point 3-11
registers 4-2, 4-4, 4-6	WEDIT fault 5-18
Initiating	addressing next 5-1
a protection fault 5-8	C/350
a transfer to another segment 5-8	equivalent 9-8, 9-11
fixed-point overflow fault 5-14	memory reference 9-5f
floating-point fault 5-15	character move 5-12
I/O operation 7-5	CPU device 7-6
Insert	determining address of fault 5-16ff
character	device flags for skip 7-5
depending on sign flag 2-21	DO-loop 5-3
depending on trigger 2-21	EDIT 9-3
j times 2-21, 10-25	equivalent 9-7, 9-9f
once 2-21, 10-25	equivalent 32-bit processor 9-7
suppress 10-25	execute accumulator 5-2
characters immediate 2-21, 10-25	
sign 10-25	

executing	executing
C/350 memory reference 9-4ff	16-bit 9-1
I/O 7-1ff	floating-point 3-4
jump 5-1	fixed-point
LEF 7-1ff	add 2-22
return 5-1	addition 2-4
skip 5-1	arithmetic 2-4
XCT 5-1	byte movement 2-20
floating-point divide 3-11	data movement 2-3
format general I/O 7-4	division 2-5
INTDS CPU device 7-6	logical 2-12
INTEN CPU device 7-6	logical skip 2-14
interrupting an 7-6	move 2-3
I/O 7-2ff, 8-3	multiplication 2-5
issuing an I/O 7-5	subtraction 2-3
load	and the second s
effective address 8-3	floating-point
	addition 3-6
physical address 7-2	arithmetic 3-6
mask out 7-6, 7-12	binary conversion 3-3
memory reference 1-8	data move 3-4
port select 7-2	decimal conversion 3-3
restarting interrupted 7-6	division 3-8
resume memory reference 8-4	move 3-3
resuming interrupted 7-6	skip on condition 3-9
setting AC0 with address of fault 8-13	status 3-10
size 1-8	general I/O 7-3
time of fault address of fault 5-14ff	hex shift 2-22
vector interrupt 7-7	initializing carry flag 2-6
wide	interrupting specific 2-11
return 5-6f, 5-13, 9-3f	jump 5-1f
save 4-3, 5-12	load
special save 5-5ff, 9-3f	effective byte address 2-21
Instructions	effective word address 2-21
for BMC maps, I/O 7-3	logical 2-13
for data channel maps, I/O 7-3	shift 2-14
to manipulate modified flag 8-11	skip on condition 2-14
to manipulate referenced flag 8-11	move decimal/floating-point 2-20
Instructions,	multi-word wide stack 4-8
byte compare 2-15	NIO B-2
byte move 2-20f	
C/350 9-4	noninterruptible 7-6
fixed-point 9-7f	privileged 8-2, 8-11 PSR manipulation 2-9
floating-point 9-9	
	queue 6-5
memory reference 9-1, 9-4ff	search queue 5-4
program flow 9-2	segment transfer 5-8
signed divided B-2	sequence of subroutine 5-6
stack 9-11	shift 2-6
stack management 9-11	skip 2-7, 2-14, 2-22f, 3-8, 5-2f
stack-referenced 9-1	subroutine 5-5
central processor identification 8-11	system identification 8-11
convert decimal/floating-point 2-20	wide F-2
decimal	arithmetic shift 2-6
arithmetic 2-22	skip on accumulator bit 2-23
fixed-point subtract 2-22	stack 4-4f
move 2-20f	stack return block 4-5
move and convert 2-20	INTDS CPU device instruction 7-6
shift 2-22	Integer extended,
DO until greater than 5-3	load 9-9, 10-69
double-word stack access 4-5	store 9-9, 10-113
EDIT subprogram 2-21	
· · · · · · · · · · · · · · · · · · ·	

wide	servicing
load 9-9, 10-136f	an 7-6
store 9-9, 10-155	base level vector 7-9, 10-144
Integer,	intermediate level vector 7-11ff, 10-143
convert to 16-bit 10-19	vector 7-3
converting a decimal 2-21	vector on I/O 4-8
load 9-9, 10-68	wide restore from an I/O 4-5, 5-8, 7-7
store 9-9, 10-112	Interrupt-executed opcode flag 2-10f
wide	Interrupted
load 9-9, 10-136	instruction 7-6
store 10-154f	program,
Integerize floating-point 3-3, 10-46f	executing 5-14
INTEN CPU device instruction 7-6	restarting 8-10
Intermediate	an instruction 7-6
	device number 7-12f
exponent,	device, vector on 10-172
decrementing an 3-5	Interrupting specific instructions 2-11
producing an 3-8	Interrupts 7-5
floating-point exponent 3-7	Interrupts,
level interrupt processing 5-8, 7-9ff, 10-143	
logical address 1-11	C/350 faults and 9-3
mantissa 3-5ff	honoring 7-13
result,	I/O 9-3
rounding the 3-11	priority of handling I/O 5-14
shifting an 2-6	Invalid
signs of the 3-5	ASCII data format 5-13
truncating 3-11	data type 6 or 7 5-17, E-2
Internal	decimal data 1-16
device 7-4	decimal data format 5-13
processor state 7-9	digit 5-17, E-2
Interpreting LEF and I/O opcodes 7-1ff	microinterrupt return block fault code E-1
Interrupt	opcode 5-17, E-2
communication 7-5	page 8-4
fault handler, first instruction of 7-13	segment 8-2
handler 7-5f, 9-3, 10-172	sign code 5-17, E-2
level word 7-9	Invoking interrupt system 7-5, 8-10
mask 7-5f, 7-12f	Inward
on flag 7-4ff	address reference fault code E-1
processing 5-6, 5-8, 7-7, 7-9, 7-11	call 5-8
recognition 7-5, 7-12	reference violation 8-13f
request,	return fault code E-1
blocking an 7-5f	return, illegal 5-9
detecting an I/O 5-1	I/O
I/O 2-10, 4-1	access, validate 8-2
	channel mask 7-12
servicing an 7-5, 8-13f	command, issuing programmed 7-3
resume flag (see IRES flag)	controller, communicating with 7-3
routine 7-12f	
routine dependent information 7-12	data transfer 7-1ff
sequence 7-8	immediate, command 7-3, 10-11
service routine 7-5	instruction 7-2ff, 8-3
service, concluding vector 7-11	instruction execution, enabling 7-1ff
system 1-5, 7-5, 8-10	instruction violation 8-13f
Interrupt,	instructions, general 7-3
intermediate level 5-8	interrupt 5-6, 5-8, 5-14, 5-20, 5-23, 9-3
I/O 5-6, 5-8, 5-14, 5-20, 5-23	interrupt recognition 7-12
I/O vector 2-9	interrupt request 2-10, 4-1
returning from I/O 4-1, 5-6, 5-8	interrupt request,
	detecting an 5-1
	servicing 8-13f

interrupt,	to subroutine,
returning from 5-6, 5-8	extended 9-10, 10-37
vector on 4-8	push and 4-5, 4-8, 5-6
wide restore from an 5-8	Jump,
interrupts, priority of handling 5-14	extended 9-10, 10-37
memory reference 8-11	pop PC and 9-10f, 10-102
mode defined 7-1ff	push 4-8, 5-5, 9-10f, 10-82, 10-103, 10-171
opcodes, interpreting LEF and 7-1ff	wide pop PC and 4-5, 4-8, 5-5, 9-10f, 10-144
operation,	V
initiating and finishing an 7-5	K
legal 8-3	Kbyte address boundary, two 8-3
port (see I/O channel)	
protection violation fault code E-1	Kernel operating system 1-6
reset 2-10, 7-3	L
skip 7-3, 10-109	L
system calls 7-5	L instruction prefix 1-11
transfer 7-2f, 10-96	Large array, expanding data areas for 9-3
	Last instruction of
validity flag 1-17, 7-2, 8-3	
vector interrupt 2-9	fault handler 5-14, 5-21, 5-23
I/O,	interrupt handler 7-7
burst multiplexor channel 1-5f, 7-1f	subroutine 5-13
command 7-3, 10-11	LCALL 2-9, 4-4ff, 4-8, 5-4ff, 10-65f
data channel 1-5f, 7-1ff	LCALL count 4-6
programmed 1-5, 7-1, 10-101	LCPID 8-11, 10-66
ION flag 7-4ff	LDA 9-8, 10-66
IOR 2-12f, 10-63	LDAFP 4-4, 10-67
IORI 2-13, 10-64	LDASB 4-4, 10-67
IORST 7-3	LDASL 4-4, 10-67
IRES flag,	LDASP 4-4, 10-67
defined 2-10f	LDATS 2-3, 4-5, 10-67
initializing 7-13	LDB 9-8, 10-68
saving state of 5-15	LDI 9-9, 10-68
setting 5-20, 7-6, 8-13, 9-3	LDIX 5-17, 9-9, 10-69, E-2
Issuing	LDSP 5-2, 9-10, 10-69f
an I/O instruction 7-5	Lead bit,
programmed I/O command 7-3	locate 2-13, 10-80
read or write command to device map 7-3	locate and reset 2-13, 10-82
ISZ 9-8, 10-64	wide
ISZTS 2-9, 4-5, 10-64	locate 2-13, 10-137
IXCT flag, defined 2-10f	locate and reset 2-13, 10-138
	Least significant
J	bit 1-2, 2-6f
	digit 2-17
J	LEF 8-3, 9-10, 10-70
alphabetical characters, move 2-21	
characters, move 2-21	and I/O opcodes, interpreting 7-1ff
float, move 2-21	instruction, executing 7-1ff
numerics, move 2-21	mode flag 8-3
times, insert character 2-21, 10-25	mode,
JMP 9-10, 10-65	defined 7-1ff, 8-3
JSR 9-4, 9-10f, 10-65	selecting 7-1ff
	Left immediate, wide logical shift 2-14
Jump 9-4, 9-10, 10-65, 10-75, 10-165	Left,
(with extended or long displacement) 5-2	DHXL double hex shift 2-22, 10-23
if nonzero, decrement and 10-21	HXL single hex shift 2-22, 10-61
indirect 3-9, 7-7, 8-10	shifting one bit to the 2-6
instructions 5-1f	Legal I/O operation 8-3
relative to program counter 9-10	
to fault handler 5-13ff, 8-13	Level
to stack fault handler 5-23	one, page 8-7
to subroutine 5-4ff, 9-3f, 9-10, 10-65, 10-76, 10-165	two, page 8-7
. , , , , , , , , , , , , , , , , , , ,	Level.

microcode revision 8-11 translation 8-3	WSL 4-4, 10-67 WSP 4-4, 10-67
Levels of indirection 5-13, 8-13f	Load
LFAMD 3-6, 10-71	accumulator,
LFAMS 3-6, 10-71	double word addressed by WSP 2-3
LFDMD 3-8, 10-71	extended 9-8, 10-38
LFDMS 3-8, 10-72	narrow 2-3, 9-8, 10-79, 10-168
LFLDD 3-4, 10-72	wide 2-3, 10-87, 10-174
LFLDS 3-4, 10-72 LFLDS 3-4, 10-72	all segment base registers 10-82f
LFLST 3-4, 3-10f, 9-9, 10-73	byte 2-20, 9-8, 10-68, 10-76, 10-165
LFMMD 3-7, 10-73	byte,
LFMMS 3-7, 10-74f	extended 9-8, 10-38
LFSMD 3-7, 10-74	wide 2-20, 9-8, 10-136
LFSMS 3-7, 10-74	CPU identification,
LFSST 3-10, 5-16, 9-9, 10-74	C/350 8-11, 10-35
LFSTD 3-4, 10-75	long 8-11, 10-66
LFSTS 3-4, 10-75	narrow 8-11, 10-93
Limit,	effective address 2-12ff, 2-21f, 8-3, 9-10, 10-70, 10-76,
bit zero of	10-165
narrow stack 5-22	effective
wide stack 5-21	address, extended 9-10, 10-39
initializing wide stack 8-10, 8-13f	byte address 1-13, 2-21, 10-76, 10-166
loading wide stack 5-11, 5-13	byte address instructions 2-21
lower stack 4-2	word address instructions 2-21
narrow stack 5-21ff, 9-2	exponent 10-45
setting bit zero of narrow stack 5-22f	floating-point
upper stack 4-2	double 9-9, 10-47, 10-72, 10-162
wide stack 4-3, 4-7, 5-11, 5-13, 5-20, 7-9, 7-11, 8-10	single 9-9, 10-47, 10-72, 10-162
zero-extending vector stack 7-9	status 9-9, 10-48, 10-73
Limits	immediate, narrow 2-3, 10-96
and skip,	in memory 2-20
compare to 9-8, 10-12	integer 9-9, 10-68
wide compare to 2-8, 9-8, 10-122	extended 9-9, 10-69
of narrow stack 9-2	extended, wide 9-9, 10-136f
of wide stack 4-2f	integer, wide 9-9, 10-136
Link 6-2	map, wide 10-137
LJMP 5-2, 7-7, 10-75	modified and referenced bits 8-11, 10-77
LJSR 5-4ff, 9-3f, 10-76	physical
LLDB 2-20, 10-76	address instruction 7-2
LLEF 2-12ff, 2-21, 10-76	and conditional skip 10-81
LLEFB 2-21, 10-76	processor status register in AC0 10-82
LMRF 8-11, 10-77	PSR in AC0 2-9f
LNADD 2-4, 10-77	sign 9-8, 10-84
LNADI 2-4, 10-77	sign, wide 2-13, 2-23, 9-8, 10-139
LNDIV 2-5, 10-78	with wide immediate, wide 10-135
LNDO 5-3, 10-78	Loading
LNDSZ 2-9, 10-78	ACO with address of fault 5-20
LNISZ 2-9, 10-79	AC1 with fault code 5-17ff, 9-3
LNLDA 2-3, 10-79	AC2 with base address of DCT 7-12
LNMUL 2-5, 10-79	byte in accumulator 2-20
LNSBI 2-4, 10-79	byte in memory 2-20
LNSTA 2-3, 10-80	device map 7-3
LNSUB 2-4, 10-80	exponent into FPAC 3-3
Load	fault code in AC1 5-17ff, 9-3
accumulator 9-8, 10-66	floating-point data 3-4
accumulator with	FPSR 3-10
double word 10-67	in FPAC, converting a decimal and 2-20
double word addressed by WSP 4-5	physical address into accumulator 7-3
WFP 4-4, 10-67	program counter 5-13
WSB 4-4, 10-67	referenced page from disk 8-10

segment base register 8-2	LSBRA 8-2, 10-82f
the vector stack 7-9	LSBRS 8-2, 10-83
wide frame pointer 5-12f	LSH 2-14, 10-84
wide stack	LSN 9-8, 10-84
base 5-11, 5-13	LSTB 2-20, 10-85
limit 5-11, 5-13	LWADD 2-4, 10-85
pointer 5-11, 5-13	LWADI 2-4, 10-85
LOB 2-13, 10-80	LWDIV 2-5, 10-85f
Locate	LWDO 5-3, 10-86
and reset lead bit 2-13, 10-82	LWDSZ 2-9, 10-86
and reset lead bit, wide 2-13, 10-138	LWISZ 2-9, 10-87
lead bit 2-13, 10-80	LWLDA 2-3, 10-87
lead bit, wide 2-13, 10-137	LWMUL 2-5, 10-87
Location of	LWSBI 2-4, 10-88
narrow stack 5-22	LWSTA 2-3, 10-88
wide stack, top 4-3	LWSUB 2-4, 10-88
Location,	.,
binary point 3-2	M
defining stack 4-2	
updating WSP reserved memory 5-20	Machine status, testing 5-2
Logic instructions 2-12f	Magnitude F-3
	Maintaining
Logical	excess 64 notation 3-7f
address 5-1, 8-2ff, 8-6, F-1	I/O facilities 7-5
address space 8-1, 8-10	Management instructions, C/350
address,	program flow 9-10
intermediate 1-11	stack 9-11
most significant bits translating 7-2	Management,
translating 1-7, 7-2f, 8-2, 8-4	device 1-5, 7-1
data formats 2-12	memory 1-6, 8-1
instructions 2-12f	program flow 1-5, 5-1
memory 8-1	queue 1-5, 6-1
negate, optional shift 2-14	stack 1-4
one's complement, optional shift 2-14	system 1-6, 8-1
operations 2-1, 2-12	Managing stack operations 4-1
shift 2-14, 10-84	Manipulating
immediate, wide 2-14, 10-139	a device flag 7-5
instructions 2-14	an interrupt mask 7-5
left immediate, wide 2-14	an interrupt on flag 7-5
with narrow immediate, wide 10-138	modified flag 8-11
shift,	referenced flag 8-11
double 2-14, 10-27	
wide 2-14, 10-138	string of bytes 2-21 instructions, PSR 2-9
skip	Manipulation, byte 2-1
instructions, fixed-point 2-14	
on condition instructions 2-14	Mantissa 3-2, 3-8
word address 8-6	divide by zero flag 3-11
Long	equals zero 3-1
displacement 1-11	overflow 3-5f
load CPU identification 8-11, 10-66	overflow flag 3-11
Lower	sign, complementing the 3-7
128 Kbyte, restricting vector stack to 7-9	status 3-9
stack limit 4-2	Mantissa,
Lower-numbered segment 4-3	adding one to the intermediate 3-6
LPEF 2-21, 4-5, 4-8, 10-81	aligning 3-4f
LPEFB 2-21, 4-5, 4-8, 10-81	final intermediate 3-7
LPHY 7-2f, 10-81	intermediate 3-5ff
LPSHJ 4-5, 4-8, 5-5f, 10-82	normalizing intermediate 3-5ff
LPSHJ, pushing with 4-5 LPSR 2-9f, 10-82	
LRB 2-13, 10-82	

range	Memory
of 3-2	access restrictions 1-6
of the intermediate 3-5	address space 1-1
rounding	address space 1-1 address translation 8-2f, 8-6
an intermediate 3-5ff	addressing 1-5, 1-8, 9-10
down an even 3-5	
the 3-4	and device, transfers between 7-1ff location,
up an odd 3-5	· · · · · · · · · · · · · · · · · · ·
shifting intermediate 3-5f	narrow skip on all bits set in 10-97f
truncating 3-4	narrow skip on any bit set in 10-98
truncating intermediate 3-6ff	updating WSP reserved 5-20
Map register assignments and format 7-2	wide skip on all bits set in 10-146
Map, device 7-2f, 10-137	wide skip on any bit set in 10-146
Mapped	management 1-6, 8-1
memory addressing 1-5	operand 1-12
mode 7-2	page, disk resident 8-1
and demand paging F-1	pages, swapping 8-7
Maps,	read operation 8-5
	reference
disabling data channel and BMC 7-3	instruction 1-8
I/O instructions for BMC 7-3	instruction,
	aborting 8-2
data channel 7-3 Mask	C/350 9-1, 9-4ff
	resume 8-4
bit 7-12	reference instructions, C/350 9-4ff
out instruction 7-6, 7-12	reference, I/O 8-11
skip and store if equal, wide 10-139f	resident
Mask,	flag 8-4
changing interrupt 7-6	page 8-4f, 8-11
current interrupt 7-12f	segment, defined 1-4
enabling OVK 4-8	size 8-11
fault service 1-3	space, physical 8-10
fixed-point overflow 2-10	word to accumulator,
FTE 3-11	narrow add 2-4, 10-77, 10-166
initializing OVK 5-15, 7-13	wide add 2-4, 10-85, 10-172
interrupt 7-5f, 7-12f	word,
I/O channel 7-12	narrow add accumulator to 2-4
loading current interrupt 7-12	narrow divide 2-5, 10-78, 10-167
manipulating an interrupt 7-5	narrow multiply 2-5, 10-79, 10-168
overflow fault service 1-2	narrow subtract 10-80, 10-169
OVK 2-10, 3-11, 4-8, 5-14, 5-20, 7-13, 8-13, 9-10f	subtract 2-4
resetting	wide divide 2-5, 10-85f, 10-173
OVK 2-9 TE 3-10	wide multiply 2-5, 10-87, 10-175
	wide subtract 10-88, 10-175f
setting	words, reserved 5-20
fixed-point overflow fault 5-14	write operation 8-5
floating-point fault 5-15	Memory,
OVK 2-9, 8-13	accessing 1-8, 8-2
TE 3-9ff, 5-15f	convert
trap enable 3-9ff, 5-15 wide	FPAC data and load in 2-20
	the four FPAC's and load in 2-20
save and reset overflow 4-5, 5-5, 10-147	defined virtual 1-6
save and set overflow 4-5, 5-5, 10-147f	fix to 9-9, 10-46
special save and reset overflow 4-5, 5-5, 9-3, 9-10f,	float from 9-9, 10-48
10-153	loading byte in 2-20
special save and set overflow 4-5, 5-5, 9-3, 9-10f,	logical 8-1
10-153f Maximum number of actor 5, 105	moving bytes in 2-15
Maximum number of gates 5-10f	narrow
	skip on all bits set in 2-8
•	skip on any bit set in 2-8
	page table in 8-1

reserved 4-1, 5-11, 5-13ff, 5-17, 7-9, 7-11, 8-7, 8-10, 8-13f, 9-2 restoring page from 8-10 virtual 8-1 vide skip on all bits set in 2-8 any bit set in 2-8 any bit set in 2-8 writing to 8-7 dicrocode revision level 8-11 dice character 9-2, 10-124 wide character 9-8, 10-136 vide character 9-8, 10-136 vide character 9-8, 10-136 vide	physical 8-1, 8-7	j
restoring page from 8-10 virtual 8-1 virtu		alphabetical characters 2-21
witual 8-1 wide skip on all bits set in 2-8 any bit set in 2-8 dicrocode revision level 8-11 dicrointerrupt E-2 or fault 5-21 return block fault code, invalid E-1 dixed number to fraction, converting 3-3 Mememonic, device address 7-4 dode, defined 1/O 7-1ff LEF 7-1ff, 8-3 mapped or unmapped 7-2 Modified and referenced bits, load 8-11, 10-177 store 8-11, 10-109 bits, table of referenced and 8-1 lag 8-11 Modify stack pointer 9-11, 10-90 pointer, wide 4-4, 5-21, 9-11, 10-141 Modifying current segment field 1-5 guard digit 3-4 segment base register 8-2 stack register contents 4-2 WSP register 4-4 MOF flag 3-11, 5-15 Moves significant bit 2-7 bit, defined 1-2 digit 2-17 bit, defined 1-2 digit 2-17 bit, defined 1-2 digit 2-17 and convert instructions, decimal 2-20 and skip 2-3 characters 10-28 decimal/floating-point instructions 2-20 digit with optional skip 2-6, 2-8, 10-48 floating-point data 3-4 MSKO 7-6, 7-12 Multiple accumulators, pop 9-8, 9-11, 10-101, 10-142 push 9-8, 9-11, 10-101, 10-142 Multiplex or channel structions, fixed-point 2-5 Multiple accumulators, pop 9-8, 10-124 Multiplex or channel structions 4-8 Multiple accumulators, pop 9-8, 10-10 block and and 9-8, 10-8f character 9-8, 10-10 block 3-9-8, 10-10 block 3-9-8, 10-10 block 3-9-8, 10-10 block 2-3, 10-140 with optional skip 2-6, 2-8, 10-88ff Move, block 9-8, 10-10 block 3-9-8, 10-10 block	8-13f, 9-2	
wirtual 8-1 numerics 2-21 wide skip on all bits set in 2-8 any bit set in 2-8 numerics 10-29 any bit set in 2-8 numerics 10-29 any bit set in 2-8 writing to 8-7 dicrocode revision level 8-11 dicrocode revision level 8-11 dicrocode revision level 8-11 dicrocode 10 dicrocode revision level 8-11 dicrocode 10 dicrocode 10 declined 10 defined d1/07-1ff LEF 7-1ff, 8-3 dod ferenced bits, load 8-11, 10-17 load 8-11, 10-17 store 8-1, 10-10 bity 4 stock pointer 9-11, 10-90 dod stage 1-1, 10-90 pointer, wide 4-4, 5-21, 9-11, 10-141 Moving Moving year 1-1, 10-10 MSP 9-1, 11, 10-10 Moving year 2-1, 10-20	restoring page from 8-10	float 2-21
all bits set in 2-8 any bit set in 2-8 any bit set in 2-8 writing to 8-7 dicrocode revision level 8-11 with circointerrupt E-2 or fault 5-21 or fault 5-22 or fault 5-22 or fault 5-22 or fault 5-23 or fault 5-23 or fault 5-24 or fault 5-25 o		
all bits set in 2-8 any bit set in 2-8 writing to 8-7 Microcode revision level 8-11 Microinterrupt E-2 or fault 5-21 or fault	wide skip on	numeric with zero suppression 2-21, 10-30
any bit set in 2-8 writing to 8-7 dicrocode revision level 8-11 dicroinerrupt E-2 or fault 5-21 return block fault code, invalid E-1 dixed number to fraction, converting 3-3 dixed mombic, device address 7-4 dode, defined 1/O 7-1ff LEF 7-1ff, 8-3 mapped or unmapped 7-2 dodified and referenced bits, load 8-11, 10-179 store 8-11, 10-109 bits, table of referenced and 8-1 flag 8-11 dodify stack pointer, wide 4-4, 5-21, 9-11, 10-141 dodify stack pointer, wide 4-4, 5-21, 9-11, 10-141 dodify stack pointer, wide 4-4, 5-1, 9-11, 10-141 dodifying current segment field 1-5 guard digit 3-4 segment base register 8-2 stack register contents 4-2 WSP register 4-4 downward wide stack instructions 4-8 dultiple accumulators, pop 9-8, 9-11, 10-102f wide pop 2-3, 4-1, 4-5, 4-8, 9-8, 9-11, 10-142 digit 2-17 downward wide stack instructions fixed-point 2-5 decimal/floating-point instructions 2-20 digit with overpunch 2-21, 10-29 float 10-28 floating-point to 5-00 instruction, character 5-12 instructions, 5 decimal 2-20 fexced and seven and seve		numerics 10-29
writing to 8-7 dicrocode revision level 8-11 dicrointerrupt E-2 or fault 5-21 return block fault code, invalid E-1 dixed number to fraction, converting 3-3 dimemonic, device address 7-4 dode, defined I/O 7-1ff LF 7-1ff, 8-3 mapped or unmapped 7-2 dodified and referenced bits, load 8-11, 10-77 store 8-11, 10-109 bits, table of referenced and 8-1 flag 8-11 Modify stack pointer 9-11, 10-90 pointer, wide 4-4, 5-21, 9-11, 10-141 Modifying current segment field 1-5 guard digit 3-4 segment base register 8-2 stack register contents 4-2 WSP register 4-4 WOF flag 3-11, 5-15 Most significant bit 2-7 bit, defined 1-2 digit 2-17 MOV 2-3, 2-6, 2-8, 10-88ff Move alphabetics 10-27 and convert instructions, decimal 2-20 and skip 2-3 downment instructions 4-2 MSKO 7-6, 7-12 MSP 9-11, 10-90 MULS 2-5, 10-91 Multi-word wide stack instructions 4-8 Multiple accumulators, pop 9-8, 9-11, 10-102 wide pop 2-3, 4-1, 4-5, 4-8, 9-8, 9-11, 10-142 multiplexor channel I/O, burst 1-5f, 7-1f Multiplication, floating-point 3-7 Multiply MOV 2-3, 2-6, 2-8, 10-88ff Move alphabetics 10-27 and convert instructions, decimal 2-20 and skip 2-3 floating-point tol-50 instruction, character 5-12 instructions, character 5-10 missigned 2-5, 10-91 unsigned 2-5, 10-91 unsigned 2-5, 10-141		right, wide 2-21, 10-140
dicroode revision level 8-11 differointerrupt E-2 or fault 5-21 return block fault code, invalid E-1 dixed number to fraction, converting 3-3 dinemonic, device address 7-4 dixed number to fraction, converting 3-3 dinemonic, device address 7-4 dixed number to fraction, converting 3-3 dinemonic, device address 7-4 dixed number to fraction, converting 3-3 dinemonic, device address 7-4 dixed number to fraction, converting 3-3 diremonic, device address 7-4 dixed number to fraction, converting 3-3 diremonic, device address 7-4 dixed number to fraction, converting 3-3 diremonic, device address 7-4 divention of the provided and self and 9-8, 10-18f wide 2-3, 10-140 block add and 9-8, 10-18f wide 2-3, 10-140 block 2-3, 9-8, 10-119f character 9-8, 10-134f with optional skip 2-6, 2-8, 10-88ff wide 2-3, 10-10 block add and 9-8, 10-8f character 9-8, 10-124f with optional skip 2-6, 2-8, 10-88ff wide 2-3, 10-140 block add and 9-8, 10-8f character 9-8, 10-124f with optional skip 2-6, 2-8, 10-88ff wide 2-3, 10-140 block add and 9-8, 10-8f character 9-8, 10-124f with optional skip 2-6, 2-8, 10-88ff wide 2-3, 10-140 block add and 9-8, 10-8f character 9-8, 10-124f with optional skip 2-6, 2-8, 10-88ff wide 2-3, 10-140 block add and 9-8, 10-8f character 9-8, 10-124f with optional skip 2-6, 2-8, 10-88ff wide 2-3, 10-140 block add and 9-8, 10-8f character 9-8, 10-124f with optional skip 2-6, 2-8, 10-88ff wide 2-3, 10-140 block add and 9-8, 10-18f wide 2-3, 10-140 block add and 9-8, 10-8f character 9-8, 10-105 wide 2-3, 10-140 block add and 9-8, 10-18f wide 2-3, 10-140 block 2-3, 9-8, 10-125f wovement instructions, fixed-point byte 2-20 decimal 1-10-109 bytes 2-15, 2-20f decimal numbers 2-15 floating-point data 3-4 MSKO 7-6, 7-12 MSP 9-11, 10-90 MUL 2-5, 10-90 MUL 2-5, 10-90 MUL 2-5, 10-91 MUL 2-5, 10-90 MUL 2-5, 10-91 Multi-word wide stack instructions 4-8 Multiple accumulators, pop 9-8, 9-11, 10-101, 10-142 wide pop 2-3, 4-1, 4-5, 4-8, 9-8, 9-11, 10-102 Multiplexor channel I/O, burst 1-5f, 7-1f Multiplication instructions, fixed-poin		
wide character 2-20, 9-8, 10-124f with optional skip 2-6, 2-8, 10-88ff whose, address 7-4 with optional skip 2-6, 2-8, 10-88ff whose, defined 1/O 7-1ff LEF 7-1ff, 8-3 mapped or unmapped 7-2 wolfied and referenced bits, load 8-11, 10-77 store 8-11, 10-109 bits, table of referenced and 8-1 flag 8-11 wolfly stack pointer 9-11, 10-90 pointer, wide 4-4, 5-21, 9-11, 10-141 wolfying current segment field 1-5 guard digit 3-4 segment base register 8-2 stack register contents 4-2 wSP register 4-4 wolf of 12-2 digit 2-17 bit, defined 1-2 digit 2-17 bit, defined 1-2 digit 2-17 and convert instructions, decimal 2-20 and skip 2-3 characters 10-28 decimal/floating-point instructions 2-20 digit with overpunch 2-21, 10-29 float 10-28 floating-point 10-50 instruction, character 5-12 instructions, byte 2-20f fixed-point 2-3 wide single 2-5, 10-91 unsigned 2-5, 10-91 unsigned 2-5, 10-141 wide signed 2	Microcode revision level 8-11	
or fault 5-21 with optional skip 2-6, 2-8, 10-88ff Move, 10de, defined 1/O 7-1ff LEF 7-1ff, 8-3 mapped or unmapped 7-2 Modified and referenced bits, load 8-11, 10-77 store 8-11, 10-109 bits, table of referenced and 8-1 flag 8-11 Modify stack pointer 9-11, 10-90 pointer, wide 4-4, 5-21, 9-11, 10-141 Modifying current segment field 1-5 guard digit 3-4 segment base register 8-2 stack register contents 4-2 WSP register 4-4 MOF flag 3-11, 5-15 Most significant bit 2-7 bit, defined 1-2 digit 2-17 MOV 2-3, 2-6, 2-8, 10-88ff Movement instructions, fixed-point 2-5 Multiplex or channel 1/O, burst 1-5f, 7-1f Multiplication, floating-point 3-7 dultiply floating-point instructions, decimal 2-20 digit with overpunch 2-21, 10-29 float 10-28 decimal/floating-point instructions, byte 2-20f decimal 2-20 digit with overpunch 2-21, 10-29 float 10-28 floating-point instructions, byte 2-20f decimal 2-20 decimal 2-20 feating-point instructions, byte 2-20f decimal 2-20 feating-point instructions, byte 2-20f decimal 2-20 ffixed-point 2-3 wide signed 2-5, 10-91 unsigned 2-5, 10-141 wide signed 2-5, 10-141 wide signed 2-5, 10-141		
return block fault code, invalid E-1 Mixed number to fraction, converting 3-3 Minemonic, device address 7-4 block 9-8, 10-10 block add and 9-8, 10-8f character 9-8, 10-15f wide 2-3, 10-140 block 2-3, 9-8, 10-119f character 2-20, 9-8, 10-125f wide 2-3, 10-140 block 2-3, 9-8, 10-119f character 2-20, 9-8, 10-125f Movement instructions, fixed-point byte 2-20 data 2-3 Moving byte 2-20 data 2-3 Moving bytes 2-15, 2-20f decimal numbers 2-15 floating-point data 3-4 MSKO 7-6, 7-12 MSP 9-11, 10-90 pointer, wide 4-4, 5-21, 9-11, 10-141 Molifying current segment field 1-5 guard digit 3-4 segment base register 8-2 stack register contents 4-2 MSF register 4-4 MOF flag 3-11, 5-15 Most significant bit 2-7 bit, defined 1-2 digit 2-17 MOV 2-3, 2-6, 2-8, 10-88ff Move alphabetics 10-27 and convert instructions, decimal 2-20 and skip 2-3 characters 10-28 decimal/floating-point instructions 2-20 digit with overpunch 2-21, 10-29 float 10-28 floating-point 10-50 instruction, character 5-12 instructions, byte 2-20f decimal 2-20f fixed-point 2-3 wide signed 2-5, 10-91 unsigned 2-5, 10-91 fixed-point 2-5 marrow under stack instructions 4-8 wide 2-5, 10-91 unsigned 2-5, 10-91 unsigned 2-5, 10-91 fixed-point 2-5 middle 2-5, 10-91 unsigned 2-5, 10-91 fixed-point 2-5 middle 2-5, 10-91 unsigned 2-5, 10-91 unsigned 2-5, 10-91 unsigned 2-5, 10-141	•	with optional skip 2-6, 2-8, 10-88ff
block 9-8, 10-10 Mode, defined 1/O 7-1ff LEF 7-1ff, 8-3 mapped or unmapped 7-2 Modified and referenced bits, load 8-11, 10-77 store 8-11, 10-109 bits, table of referenced and 8-1 flag 8-11 Modify stack pointer 9-11, 10-90 pointer, wide 4-4, 5-21, 9-11, 10-141 Modifying current segment field 1-5 guard digit 3-4 segment base register 8-2 stack register contents 4-2 WSP register 4-4 MOF flag 3-11, 5-15 Most significant bit 2-7 bit, defined 1-2 digit 2-17 Move alphabetics 10-27 and convert instructions, decimal 2-20 and skip 2-3 characters 10-28 floating-point instructions 2-20 digit with overpunch 2-21, 10-29 float 10-28 floating-point instructions, decimal 2-20 digit with overpunch 2-21, 10-29 float 10-28 floating-point instructions, sylve 2-206 decimal 7-00-97 narrow sign extend 2-5 signed 2-5, 10-90 sinstruction, character 5-12 instructions, byte 2-206 decimal 7-00-97 narrow sign extend 2-5 signed 2-5, 10-90 sinstructions, byte 2-206 decimal 7-00-97 narrow sign extend 2-5 signed 2-5, 10-91 unsigned 2-5, 10-141	return block fault code, invalid E-1	,
Mode, defined 1/O 7-1ff LEF 7-1ff, 8-3 mapped or unmapped 7-2 Modified and referenced bits, load 8-11, 10-77 store 8-11, 10-109 bits, table of referenced and 8-1 flag 8-11 Modifying current segment field 1-5 guard digit 3-4 segment base register 8-2 stack register contents 4-2 WSP register 4-4 MOF flag 3-11, 5-15 MOF digit 2-7 bit, defined 1-2 digit 2-17 MOV 2-3, 2-6, 2-8, 10-88ff Mowe alphabetics 10-27 and convert instructions, decimal 2-20 and skip 2-3 characters 10-28 decimal/floating-point instructions 2-20 digit with overpunch 2-21, 10-29 float 10-28 floating-point 10-50 instructions, character 5-12 instructions, byte 2-20f decimal 2-20f fixed-point 2-3 block add and 9-8, 10-8f wide 2-3, 10-140 block 2-3, 10-126 Movement instructions, fixed-point byte 2-20f decimal 1, 10-109 boths, table of referenced and 8-1 fload and 9-8, 10-15f wide 2-3, 10-140 block 2-3, 10-140 block 2-3, 10-126 Movement instructions, fixed-point byte 2-20f decimal 2-1, 10-90 boths, table of referenced and 8-1 fload and 9-8, 10-15f wide 2-3, 10-140 block 2-3, 10-140 block 2-3, 10-140 block 2-3, 10-126 Movement instructions, fixed-point byte 2-20f decimal 1-20f fixed-point 2-3 block and and 9-8, 10-119 block 2-3, 10-126 Movement instructions, fixed-point byte 2-20f decimal 2-20f block 2-3, 10-126 Movement instructions, fixed-point byte 2-20f decimal 2-20f block 2-3, 10-126 Movement instructions, fixed-point byte 2-20f decimal 2-20f block 2-3, 10-125 Movement instructions, fixed-point byte 2-20f decimal 2-20f block 2-3, 10-125 Movement instructions, fixed-point byte 2-20f decimal a-2-3 Moving byte 2-20f decimal 1-2-20f bytes 2-15, 2-20f decimal 2-20f decimal 2-20f block add and 9-8, 10-125 floateracter 2-20, 9-8, 10-125 floating-point data 3-4 Moving byte 2-20f decimal 1-2-20f bytes 2-15, 2-20f decimal 2-1, 10-129 decimal numbers 2-15 floating-point data 3-4 MSKO 7-6, 7-12 Moving byte 2-15 floating-point data 3-4 Most ignificant byte 2-2-10 decima		
Character 9-8, 10-15f		block add and 9-8, 10-8f
block 2-3, 9-8, 10-119f LEF 7-1ff, 8-3 mapped or unmapped 7-2 Modified and referenced bits, load 8-11, 10-77 store 8-11, 10-109 bits, table of referenced and 8-1 flag 8-11 Modify stack pointer 9-11, 10-90 pointer, wide 4-4, 5-21, 9-11, 10-141 Modifying current segment field 1-5 guard digit 3-4 segment base register 8-2 stack register contents 4-2 WSP register 4-4 MOF flag 3-11, 5-15 Most significant bit 2-7 bit, defined 1-2 digit 2-17 MOV 2-3, 2-6, 2-8, 10-88ff Move alphabetics 10-27 and convert instructions, decimal 2-20 and skip 2-3 characters 10-28 floating-point 10-50 instruction, character 5-12 instructions, byte 2-20f decimal 2-20f fixed-point 2-3 block 2-3, 9-8, 10-119f character 2-20, 9-8, 10-125f Movement instructions, fixed-point byte 2-20f Movement instructions, fixed-point 1-5 Movement instructions, fixed-point 1-5 Multiple accumulators, popple 4-9-11, 10-10, 10-142 multiple accumulators, popple 4-9-11, 10-10, 10-142 multiple accumulators, pop 9-8, 9-11, 10-10, 10-142 multiple accumulators, popple accumula	Mode,	
LEF 7-1ff, 8-3 mapped or unmapped 7-2 Modified and referenced bits, load 8-11, 10-77 store 8-11, 10-109 bits, table of referenced and 8-1 flag 8-11 Modify stack pointer, wide 4-4, 5-21, 9-11, 10-141 Modifying current segment field 1-5 guard digit 3-4 segment base register 8-2 stack register contents 4-2 WSP register 4-4 MOF flag 3-11, 5-15 Most significant bit 2-7 bit, defined 1-2 digit 2-17 digit 2-17 MOV 2-3, 2-6, 2-8, 10-88ff Move alphabetics 10-27 and convert instructions, decimal 2-20 and skip 2-3 characters 10-28 floating-point 10-50 instructions, haracter 5-12 instructions, byte 2-20f decimal numbers 2-15 floating-point data 3-4 MSKO 7-6, 7-12 MSP 9-11, 10-90 MUIL 2-5, 10-90f MUIL 2-5, 10-90f MUIL S 2-5, 10-90f MUIL S 2-5, 10-90f MUIL S 2-5, 10-90f Multiple protection violations 4-8 Multiple protection violations 8-12 Multiplication instructions, fixed-point 2-5 Multiplication, floating-point 3-7 Multiplication, floating-point 3-7 Multiply Mouble (FPAC by FPAC) 3-7, 10-49f double (FPAC by FPAC) 3-7, 10-168 wide 2-5, 10-87, 10-175 single (FPAC by memory) 9-9, 10-49, 10-74f, 10-163 Multiply, narrow 10-97 narrow sign extend 2-5 signed 2-5, 10-90f wide 2-5, 10-141 wide signed 2-5, 10-141	defined	wide 2-3, 10-140
LEF 7-1ff, 8-3	I/O 7-1ff	
mapped or unmapped 7-2 Modified and referenced bits, load 8-11, 10-77 store 8-11, 10-109 bits, table of referenced and 8-1 flag 8-11 Modify stack pointer 9-11, 10-90 pointer, wide 4-4, 5-21, 9-11, 10-141 Modifying current segment field 1-5 guard digit 3-4 segment base register 8-2 stack register contents 4-2 WSP register 4-4 MOF flag 3-11, 5-15 Most significant bit 2-7 bit, defined 1-2 digit 2-17 MOV 2-3, 2-6, 2-8, 10-88ff Move alphabetics 10-27 and convert instructions, decimal 2-20 and skip 2-3 characters 10-28 decimal/floating-point instructions 2-20 digit with overpunch 2-21, 10-29 float 10-28 floating-point 10-50 instructions, byte 2-20f decimal 2-20f fixed-point 2-3 Moving bytes 2-15, 2-20f decimal anumbers 2-15 floataing-point data 3-4 Mosk 7-6, 7-12 MSKO 7-6, 7-12 MSKO 7-6, 7-12 MSKO 7-6, 7-12 MSKO 7-6, 7-12 MOWILS 2-5, 10-90f MULS 2-5, 10-90f MULS 2-5, 10-90f MULS 2-5, 10-90f MULS 2-5, 10-90f Multiple accumulators, pop 9-8, 9-11, 10-101, 10-142 push 9-8, 9-11, 10-101,		
Modified and referenced bits, load 8-11, 10-109 bits, table of referenced and 8-1 flag 8-11 Modify stack pointer 9-11, 10-90 pointer, wide 4-4, 5-21, 9-11, 10-141 Modifying current segment field 1-5 guard digit 3-4 segment base register 8-2 stack register contents 4-2 WSP register 4-4 MOF flag 3-11, 5-15 Mot significant bit 2-7 bit, defined 1-2 digit 2-17 MOV 2-3, 2-6, 2-8, 10-88ff Move alphabetics 10-27 and convert instructions, decimal 2-20 and skip 2-3 characters 10-28 decimal/floating-point instructions 2-20 digit with overpunch 2-21, 10-29 float 10-28 floating-point 10-50 instruction, character 5-12 instructions, byte 2-20f decimal 2-20f fixed-point 2-3 Moving bytes 2-15, 2-20f decimal anumbers 2-15 floating-point data 3-4 MSKO 7-6, 7-12 MSKO 7-6, 7-10 MSKO 3-1, 10-10, 10-10 MUL1ple accumulators, pop 9-8, 9-11, 10		Movement instructions, fixed-point
and referenced bits,	Modified	byte 2-20
load 8-11, 10-77 store 8-11, 10-109 bits, table of referenced and 8-1 flag 8-11 Modify stack pointer 9-11, 10-90 pointer, wide 4-4, 5-21, 9-11, 10-141 Modifying current segment field 1-5 guard digit 3-4 segment base register 8-2 stack register contents 4-2 WSP register 4-4 MOF flag 3-11, 5-15 Most significant bit 2-7 bit, defined 1-2 digit 2-17 MOV 2-3, 2-6, 2-8, 10-88ff Move alphabetics 10-27 and convert instructions, decimal 2-20 and skip 2-3 characters 10-28 decimal/floating-point instructions 2-20 digit with overpunch 2-21, 10-29 float 10-28 floating-point 2-3 bits, table of referenced and 8-1 floating-point data 3-4 MSKO 7-6, 7-12 MSP 9-11, 10-90 MUL 2-5, 10-90 Multiple accumulators, pop 9-8, 9-11, 10-101, 10-142 push 9-8, 9-11, 10-102, 10-142 Multiple protection violations 8-12 Multiplexor channel 1/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 double (FPAC by memory) 9-9, 10-49, 10-73f, 10-77, 10-163 memory word, narrow 2-5, 10-79, 10-168 wide 2-5, 10-87, 10-175 single (FPAC by memory) 9-9, 10-49, 10-74f, 10-163 Multiply, narrow 10-97 narrow sign extend 2-5 signed 2-5, 10-91 unsigned 2-5, 10-90 wide 2-5, 10-141 wide signed 2-5, 10-141		data 2-3
store 8-11, 10-109 bits, table of referenced and 8-1 flag 8-11 Modify stack pointer 9-11, 10-90 pointer, wide 4-4, 5-21, 9-11, 10-141 Modifying current segment field 1-5 guard digit 3-4 segment base register 8-2 stack register contents 4-2 WSP register 4-4 MOF flag 3-11, 5-15 Most significant bit 2-7 bit, defined 1-2 digit 2-17 MOV 2-3, 2-6, 2-8, 10-88ff Move alphabetics 10-27 and convert instructions, decimal 2-20 and skip 2-3 characters 10-28 decimal/floating-point instructions 2-20 digit with overpunch 2-21, 10-29 float 10-28 floating-point 10-50 instructions, byte 2-20f decimal 2-20f fixed-point 2-3 bytes 2-15, 2-20f decimal numbers 2-15 floating-point data 3-4 MSKO 7-6, 7-12 MSP 9-11, 10-90 MULS 2-5, 10-90 MULS 2-5, 10-90 MULS 2-5, 10-90 MULS 2-5, 10-90 Multi-word wide stack instructions 4-8 Multiple accumulators, pop 9-8, 9-11, 10-101, 10-142 push 9-8, 9-11, 10-102, 10-142 Multiple protection violations 8-12 Multiplexor channel 1/0, burst 1-5f, 7-1f Multiplication instructions, floating-point 3-7 Multiply double (FPAC by FPAC) 3-7, 10-49f double (FPAC by memory) 9-9, 10-49, 10-73f, 10-77. 10-163 memory word, narrow 2-5, 10-79, 10-168 wide 2-5, 10-87, 10-175 single (FPAC by memory) 9-9, 10-49, 10-74f, 10-163 Multiply, narrow 10-97 narrow sign extend 2-5 signed 2-5, 10-90 wide 2-5, 10-91 unsigned 2-5, 10-90 wide 2-5, 10-141 wide signed 2-5, 10-141		Moving
bits, table of referenced and 8-1 flag 8-11 Modify stack pointer 9-11, 10-90 pointer, wide 4-4, 5-21, 9-11, 10-141 Modifying current segment field 1-5 guard digit 3-4 segment base register 8-2 stack register contents 4-2 WSP register 4-4 MOF flag 3-11, 5-15 Most significant bit 2-7 bit, defined 1-2 digit 2-17 Move alphabetics 10-27 and convert instructions, decimal 2-20 and skip 2-3 characters 10-28 decimal numbers 2-15 floating-point data 3-4 MSKO 7-6, 7-12 MSP 9-11, 10-90 MUL 2-5, 10-90 Multi-sect instructions 4-8 Multi-word wide stack instructions 4-8 Multiple accumulators, pop 9-8, 9-11, 10-101, 10-142 push 9-8, 9-11, 10-101, 10-142 push 9-8, 9-11, 10-101 flog wide pop 2-3, 4-1, 4-5, 4-8, 9-8, 9-11, 10-142 Multiplexor channel I/O, burst 1-5f, 7-1f Multiplication instructions, fixed-point 2-5 Multiply double (FPAC by FPAC) 3-7, 10-49f double (FPAC by memory) 9-9, 10-49, 10-73f, 10-77, 10-163 memory word, narrow 2-5, 10-79, 10-168 wide 2-5, 10-87, 10-175 single (FPAC by memory) 9-9, 10-49, 10-74f, 10-163 Multiply, narrow 10-97 narrow sign extend 2-5 signed 2-5, 10-91 unsigned 2-5, 10-90f wide 2-5, 10-141 wide signed 2-5, 10-141		bytes 2-15, 2-20f
flag 8-11 Modify stack pointer 9-11, 10-90 pointer, wide 4-4, 5-21, 9-11, 10-141 Modifying current segment field 1-5 guard digit 3-4 segment base register 8-2 stack register contents 4-2 WSP register contents 4-2 WSP register 4-4 MOF flag 3-11, 5-15 Most significant bit 2-7 bit, defined 1-2 digit 2-17 MOV 2-3, 2-6, 2-8, 10-88ff Move alphabetics 10-27 and convert instructions, decimal 2-20 and skip 2-3 characters 10-28 decimal/floating-point instructions 2-20 digit with overpunch 2-21, 10-29 float 10-28 floating-point 10-50 instructions, byte 2-20f decimal 2-20f fixed-point 2-3 floating-point data 3-4 MSKO 7-6, 7-12 MULt 2-5, 10-90 MULt 2-5, 10-90 Multi-word wide stack instructions 4-8 Multiple accumulators, pop 9-8, 9-11, 10-101, 10-142 push 9-8, 9-11, 10-101, 10-142 push 9-8, 9-11, 10-102 f wide pop 2-3, 4-1, 4-5, 4-8, 9-8, 9-11, 10-102 Multiple accumulators, pop 9-8, 9-11, 10-101, 10-142 push 9-8, 9-11, 10-101, 10-142 push 9-8, 9-11, 10-101, 10-142 push 9-8, 9-11, 10-102 f Multiple accumulators, pop 9-8, 9-11, 10-101, 10-142 push 9-8, 9-11, 10-102 f Multiple accumulators, pop 9-8, 9-11, 10-101, 10-142 push 9-8, 9-11, 10-102 f Multiple accumulators, pop 9-8, 9-11, 10-102 f Multiple accumulators, pop 9-8, 9-11, 10-101, 10-142 push 9-8, 9-11, 10-102 f Multiple accumulators, pop 9-8, 9-11, 10-101, 10-142 push 9-8, 9-11, 10-102 f Multiple accumulators, pop 9-8, 9-11, 10-102 f Multiple accumulators, pop 9-8, 9-11, 10-102 f Multiple accumulators, pop 9-8, 9-11, 10-102, 10-142 push 9-8, 9-11, 10-102 f Multiple accumulators, pop 9-8, 9-11, 10-102 f Mul		
Modify stack pointer 9-11, 10-90 pointer, wide 4-4, 5-21, 9-11, 10-141 MSP 9-11, 10-90 MUL 2-5, 10-90f MULS 2-5, 10-91 MULS 2-5, 10-91 Multi-word wide stack instructions 4-8 Multiple accumulators, pop 9-8, 9-11, 10-101, 10-142 push 9-8, 9-11, 10-101, 10-142 push 9-8, 9-11, 10-101, 10-142 push 9-8, 9-11, 10-102f wide pop 2-3, 4-1, 4-5, 4-8, 9-8, 9-11, 10-102f wide pop 2-3, 4-1, 4-5, 4-8, 9-8, 9-11, 10-142 Multiple protection violations 8-12 Multiple protection violations 8-12 Multiple protection violation instructions, fixed-point 2-5 Multiply double (FPAC by FPAC) 3-7, 10-49f double (FPAC by memory) 9-9, 10-49, 10-73f, 10-77, 10-163 memory word, narrow 2-5, 10-79, 10-168 wide 2-5, 10-87, 10-175 single (FPAC by memory) 9-9, 10-49, 10-74f, 10-163 marrow 10-97 instruction, character 5-12 instructions, byte 2-20f decimal 2-20f fixed-point 2-3 wide signed 2-5, 10-91 unsigned 2-5, 10-91 fixed-point 2-3	•	
pointer 9-11, 10-90 pointer, wide 4-4, 5-21, 9-11, 10-141 Modifying current segment field 1-5 guard digit 3-4 segment base register 8-2 stack register contents 4-2 WSP register 4-4 MOF flag 3-11, 5-15 Most significant bit 2-7 bit, defined 1-2 digit 2-17 MOV 2-3, 2-6, 2-8, 10-88ff Move alphabetics 10-27 and convert instructions, decimal 2-20 and skip 2-3 characters 10-28 decimal/floating-point instructions 2-20 digit with overpunch 2-21, 10-29 float 10-28 floating-point 10-50 instructions, byte 2-20f decimal 2-20f fixed-point 2-3 MSP 9-11, 10-90 MUL 2-5, 10-90f Multiple accumulators, Multiple accumulators, pop 9-8, 9-11, 10-101, 10-142 push 9-8, 9-11, 10-102, wide pop 2-3, 4-1, 4-5, 4-8, 9-8, 9-11, 10-142 Multiple protection violations 8-12 Multiple protection violations 8-12 Multiple protection violations 8-12 Multiplication instructions, fixed-point 2-5 Multiplication, floating-point 3-7 Multiply double (FPAC by FPAC) 3-7, 10-49f double (FPAC by memory) 9-9, 10-49, 10-73f, 10-77, 10-163 memory word, narrow 2-5, 10-87, 10-168 wide 2-5, 10-87, 10-175 single (FPAC by memory) 9-9, 10-49, 10-74f, 10-163 Multiple, narrow 10-97 narrow sign extend 2-5 signed 2-5, 10-91 unsigned 2-5, 10-90f wide 2-5, 10-141 wide signed 2-5, 10-141	Modify stack	MSKO 7-6, 7-12
pointer, wide 4-4, 5-21, 9-11, 10-141 Modifying		
Modifying current segment field 1-5 guard digit 3-4 segment base register 8-2 stack register contents 4-2 WSP register 4-4 MOF flag 3-11, 5-15 Most significant bit 2-7 bit, defined 1-2 digit 2-17 MOV 2-3, 2-6, 2-8, 10-88ff Move alphabetics 10-27 and convert instructions, decimal 2-20 and skip 2-3 characters 10-28 decimal/floating-point instructions 2-20 digit with overpunch 2-21, 10-29 float 10-28 floating-point 10-50 instructions, byte 2-20f decimal 2-20f fixed-point 2-3 MULS 2-5, 10-91 Multiple word weld estack instructions 4-8 Multiple accumulators, pop 9-8, 9-11, 10-101, 10-142 Multiple protection violations 8-12 Multiple protection violations 8-12 Multiplication instructions, fixed-point 2-5 Multiplication instructions, fixed-point 3-7 Multiply, double (FPAC by FPAC) 3-7, 10-49f double (FPAC by FPAC) 3-7, 10-49f double (FPAC by FPAC) 3-7, 10-168 wide 2-5, 10-87, 10-10-101, 1		
guard digit 3-4 segment base register 8-2 stack register contents 4-2 WSP register 4-4 MOF flag 3-11, 5-15 Most significant bit 2-7 bit, defined 1-2 digit 2-17 MOV 2-3, 2-6, 2-8, 10-88ff Move alphabetics 10-27 and convert instructions, decimal 2-20 and skip 2-3 characters 10-28 decimal/floating-point instructions 2-20 float 10-28 floating-point 10-50 instructions, byte 2-20f decimal 2-20f fixed-point 2-3 Multiple accumulators, pop 9-8, 9-11, 10-101, 10-142 push 9-8, 9-11, 10-102f wide pop 2-3, 4-1, 4-5, 4-8, 9-8, 9-11, 10-142 Multiple protection violations 8-12 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 double (FPAC by memory) 9-9, 10-49, 10-73f, 10-77, 10-163 memory word, narrow 2-5, 10-87, 10-168 wide 2-5, 10-87, 10-175 single (FPAC by FPAC) 3-7, 10-50 single (FPAC by memory) 9-9, 10-49, 10-74f, 10-163 Multiply, narrow 10-97 narrow sign extend 2-5 signed 2-5, 10-91 unsigned 2-5, 10-91 unsigned 2-5, 10-90f wide 2-5, 10-141 wide signed 2-5, 10-141	Modifying	
guard digit 3-4 segment base register 8-2 stack register contents 4-2 WSP register 4-4 MOF flag 3-11, 5-15 Most significant bit 2-7 bit, defined 1-2 digit 2-17 MOV 2-3, 2-6, 2-8, 10-88ff Move alphabetics 10-27 and convert instructions, decimal 2-20 and skip 2-3 characters 10-28 decimal/floating-point instructions 2-20 digit with overpunch 2-21, 10-29 float 10-28 floating-point 10-50 instructions, byte 2-20f decimal 2-20f fixed-point 2-3 Multiple accumulators, pop 9-8, 9-11, 10-102 f wide pop 2-3, 4-1, 4-5, 4-8, 9-8, 9-11, 10-142 Multiple protection violations 8-12 Multiplexor channel I/O, burst 1-5f, 7-1f Multiplexor channel I/O, burst 1-5f, 7-1f Multiplication instructions, fixed-point 3-7 Multiply double (FPAC by FPAC) 3-7, 10-49f double (FPAC by memory) 9-9, 10-49, 10-73f, 10-77. 10-163 memory word, narrow 2-5, 10-79, 10-168 wide 2-5, 10-87, 10-175 single (FPAC by FPAC) 3-7, 10-50 single (FPAC by memory) 9-9, 10-49, 10-74f, 10-163 Multiply, narrow 10-97 narrow sign extend 2-5 signed 2-5, 10-90f wide 2-5, 10-91f wide 2-5, 10-141 wide signed 2-5, 10-141	current segment field 1-5	
segment base register 8-2 stack register contents 4-2 WSP register 4-4 MOF flag 3-11, 5-15 Most significant bit 2-7 bit, defined 1-2 digit 2-17 MOV 2-3, 2-6, 2-8, 10-88ff Move alphabetics 10-27 and convert instructions, decimal 2-20 and skip 2-3 characters 10-28 decimal/floating-point instructions 2-20 digit with overpunch 2-21, 10-29 float 10-28 floating-point 10-50 instructions, byte 2-20f decimal 2-20f fixed-point 2-3 pop 9-8, 9-11, 10-101, 10-142 push 9-8, 9-11, 10-102 wide pop 2-3, 4-1, 4-5, 4-8, 9-8, 9-11, 10-142 Multiple protection violations 8-12 Multiplexor channel I/O, burst 1-5f, 7-1f Multiplexor ch		
WSP register 4-4 MOF flag 3-11, 5-15 Most significant bit 2-7 bit, defined 1-2 digit 2-17 MOV 2-3, 2-6, 2-8, 10-88ff Move alphabetics 10-27 and convert instructions, decimal 2-20 and skip 2-3 characters 10-28 decimal/floating-point instructions 2-20 digit with overpunch 2-21, 10-29 float 10-28 floating-point 10-50 instructions, byte 2-20f decimal 2-20f fixed-point 2-3 wide pop 2-3, 4-1, 4-5, 4-8, 9-8, 9-11, 10-142 Multiple protection violations 8-12 Multiple protection violations 4-15, 7-1f Multiple protection violations 4-15, 7-1f Multiple protection violations 4-25, 10-49f double (FPAC by FPAC) 3-7, 10-49f double (FPAC by	segment base register 8-2	
MOF flag 3-11, 5-15 Most significant bit 2-7 bit, defined 1-2 digit 2-17 MOV 2-3, 2-6, 2-8, 10-88ff Move alphabetics 10-27 and convert instructions, decimal 2-20 and skip 2-3 characters 10-28 decimal/floating-point instructions 2-20 digit with overpunch 2-21, 10-29 float 10-28 floating-point 10-50 instructions, byte 2-20f decimal 2-20f fixed-point 2-3 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 double (FPAC by memory) 9-9, 10-49, 10-73f, 10-77, 10-163 memory word, narrow 2-5, 10-79, 10-168 wide 2-5, 10-87, 10-175 single (FPAC by FPAC) 3-7, 10-50 single (FPAC by memory) 9-9, 10-49, 10-74f, 10-163 Multiply, narrow 10-97 narrow sign extend 2-5 signed 2-5, 10-91 unsigned 2-5, 10-90f wide 2-5, 10-141 wide signed 2-5, 10-141	stack register contents 4-2	
Most significant bit 2-7 bit, defined 1-2 digit 2-17 MOV 2-3, 2-6, 2-8, 10-88ff Move alphabetics 10-27 and convert instructions, decimal 2-20 and skip 2-3 characters 10-28 decimal/floating-point instructions 2-20 digit with overpunch 2-21, 10-29 float 10-28 floating-point 10-50 instructions, byte 2-20f decimal 2-20f fixed-point 2-3 Multiplication instructions, fixed-point 2-5 Multiplication, floating-point 3-7 Multiply double (FPAC by FPAC) 3-7, 10-49f double (FPAC by memory) 9-9, 10-49, 10-73f, 10-77 10-163 memory word, narrow 2-5, 10-79, 10-168 wide 2-5, 10-87, 10-175 single (FPAC by FPAC) 3-7, 10-50 single (FPAC by memory) 9-9, 10-49, 10-74f, 10-163 Multiply narrow 2-5, 10-79, 10-168 wide 2-5, 10-87, 10-175 single (FPAC by memory) 9-9, 10-49, 10-74f, 10-163 Multiply narrow 2-5, 10-90 single (FPAC by memory) 9-9, 10-49, 10-74f, 10-163 Multiply narrow 2-5, 10-91 unsigned 2-5, 10-91 unsigned 2-5, 10-91 unsigned 2-5, 10-141 wide signed 2-5, 10-141	WSP register 4-4	
bit 2-7 bit, defined 1-2 digit 2-17 MOV 2-3, 2-6, 2-8, 10-88ff Move alphabetics 10-27 and convert instructions, decimal 2-20 and skip 2-3 characters 10-28 decimal/floating-point instructions 2-20 digit with overpunch 2-21, 10-29 float 10-28 floating-point 10-50 instructions, byte 2-20f decimal 2-20f fixed-point 2-3 Multiplication instructions, fixed-point 2-5 Multiplication, floating-point 3-7 Multiply double (FPAC by FPAC) 3-7, 10-49f double (FPAC by memory) 9-9, 10-49, 10-73f, 10-77, 10-163 memory word, narrow 2-5, 10-79, 10-168 wide 2-5, 10-87, 10-175 single (FPAC by FPAC) 3-7, 10-50 single (FPAC by FPAC) 3-7, 10-50 single (FPAC by FPAC) 3-7, 10-163 Multiply, narrow 10-97 narrow sign extend 2-5 signed 2-5, 10-91 unsigned 2-5, 10-91 unsigned 2-5, 10-91 side 2-5, 10-141 wide signed 2-5, 10-141	MOF flag 3-11, 5-15	
bit, defined 1-2 digit 2-17 MOV 2-3, 2-6, 2-8, 10-88ff Move alphabetics 10-27 and convert instructions, decimal 2-20 and skip 2-3 characters 10-28 decimal/floating-point instructions 2-20 digit with overpunch 2-21, 10-29 float 10-28 floating-point 10-50 instructions, byte 2-20f decimal 2-20f fixed-point 2-3 Multiply double (FPAC by FPAC) 3-7, 10-49f double (FPAC by memory) 9-9, 10-49, 10-73f, 10-77 10-163 memory word, narrow 2-5, 10-79, 10-168 wide 2-5, 10-87, 10-175 single (FPAC by FPAC) 3-7, 10-50 single (FPAC by memory) 9-9, 10-49, 10-74f, 10-163 Multiply, narrow 10-97 narrow sign extend 2-5 signed 2-5, 10-91 unsigned 2-5, 10-90f wide 2-5, 10-141 wide signed 2-5, 10-141	Most significant	
digit 2-17 MOV 2-3, 2-6, 2-8, 10-88ff Move alphabetics 10-27 and convert instructions, decimal 2-20 and skip 2-3 characters 10-28 decimal/floating-point instructions 2-20 digit with overpunch 2-21, 10-29 float 10-28 floating-point 10-50 instructions, byte 2-20f decimal 2-20f fixed-point 2-3 Multiply double (FPAC by FPAC) 3-7, 10-49f double (FPAC by memory) 9-9, 10-49, 10-73f, 10-77 and convert instructions, decimal 2-20 and skip 2-3 characters 10-28 wide 2-5, 10-87, 10-175 single (FPAC by FPAC) 3-7, 10-50 single (FPAC by FPAC) 3-7, 10-50 single (FPAC by FPAC) 3-7, 10-168 wide 2-5, 10-87, 10-175 single (FPAC by memory) 9-9, 10-49, 10-74f, 10-163 Multiply, narrow 10-97 narrow sign extend 2-5 signed 2-5, 10-91 unsigned 2-5, 10-91 unsigned 2-5, 10-90f wide 2-5, 10-141 wide signed 2-5, 10-141	bit 2-7	
MOV 2-3, 2-6, 2-8, 10-88ff Move alphabetics 10-27 and convert instructions, decimal 2-20 and skip 2-3 characters 10-28 decimal/floating-point instructions 2-20 digit with overpunch 2-21, 10-29 float 10-28 floating-point 10-50 instructions, byte 2-20f decimal 2-20f fixed-point 2-3 double (FPAC by FPAC) 3-7, 10-49f double (FPAC by memory) 9-9, 10-49, 10-73f, 10-77f, 10-163 memory word, narrow 2-5, 10-79, 10-168 wide 2-5, 10-87, 10-175 single (FPAC by FPAC) 3-7, 10-50 single (FPAC by memory) 9-9, 10-49, 10-74f, 10-163 Multiply, narrow 10-97 narrow sign extend 2-5 signed 2-5, 10-91 unsigned 2-5, 10-90f wide 2-5, 10-141 wide signed 2-5, 10-141	bit, defined 1-2	· · · · · · · · · · · · · · · · · · ·
Move alphabetics 10-27 and convert instructions, decimal 2-20 and skip 2-3 characters 10-28 decimal/floating-point instructions 2-20 digit with overpunch 2-21, 10-29 float 10-28 floating-point 10-50 instructions, byte 2-20f decimal 2-20f fixed-point 2-3 double (FPAC by memory) 9-9, 10-49, 10-73f, 10-77, 10-163 memory word, narrow 2-5, 10-79, 10-168 wide 2-5, 10-87, 10-175 single (FPAC by FPAC) 3-7, 10-50 single (FPAC by memory) 9-9, 10-49, 10-74f, 10-163 Multiply, narrow 10-97 narrow sign extend 2-5 signed 2-5, 10-91 unsigned 2-5, 10-90f wide 2-5, 10-141 wide signed 2-5, 10-141		
alphabetics 10-27 and convert instructions, decimal 2-20 and skip 2-3 characters 10-28 decimal/floating-point instructions 2-20 digit with overpunch 2-21, 10-29 float 10-28 floating-point 10-50 instructions, byte 2-20f decimal 2-20f fixed-point 2-3 and convert instructions, decimal 2-20 memory word, narrow 2-5, 10-79, 10-168 wide 2-5, 10-87, 10-175 single (FPAC by FPAC) 3-7, 10-50 single (FPAC by memory) 9-9, 10-49, 10-74f, 10-163 Multiply, narrow 10-97 narrow sign extend 2-5 signed 2-5, 10-91 unsigned 2-5, 10-90f wide 2-5, 10-141 wide signed 2-5, 10-141	MOV 2-3, 2-6, 2-8, 10-88ff	double (FPAC by FPAC) 3-7, 10-49f
and convert instructions, decimal 2-20 and skip 2-3 characters 10-28 decimal/floating-point instructions 2-20 digit with overpunch 2-21, 10-29 float 10-28 floating-point 10-50 instructions, byte 2-20f decimal 2-20f fixed-point 2-3 memory word, narrow 2-5, 10-79, 10-168 wide 2-5, 10-87, 10-175 single (FPAC by FPAC) 3-7, 10-50 single (FPAC by memory) 9-9, 10-49, 10-74f, 10-163 Multiply, narrow 10-97 narrow sign extend 2-5 signed 2-5, 10-91 unsigned 2-5, 10-90f wide 2-5, 10-141 wide signed 2-5, 10-141	Move	
and skip 2-3 characters 10-28 decimal/floating-point instructions 2-20 digit with overpunch 2-21, 10-29 float 10-28 floating-point 10-50 instructions, byte 2-20f decimal 2-20f fixed-point 2-3 anarrow 2-5, 10-79, 10-168 wide 2-5, 10-87, 10-175 single (FPAC by FPAC) 3-7, 10-50 single (FPAC by memory) 9-9, 10-49, 10-74f, 10-163 Multiply, narrow 10-97 narrow sign extend 2-5 signed 2-5, 10-91 unsigned 2-5, 10-90f wide 2-5, 10-141 wide signed 2-5, 10-141		
characters 10-28 decimal/floating-point instructions 2-20 digit with overpunch 2-21, 10-29 float 10-28 floating-point 10-50 instructions, byte 2-20f decimal 2-20f fixed-point 2-3 wide 2-5, 10-87, 10-175 single (FPAC by FPAC) 3-7, 10-50 single (FPAC by memory) 9-9, 10-49, 10-74f, 10-163 Multiply, narrow 10-97 narrow sign extend 2-5 signed 2-5, 10-91 unsigned 2-5, 10-90f wide 2-5, 10-141 wide signed 2-5, 10-141	and convert instructions, decimal 2-20	
decimal/floating-point instructions 2-20 digit with overpunch 2-21, 10-29 float 10-28 floating-point 10-50 instruction, character 5-12 instructions, byte 2-20f decimal 2-20f fixed-point 2-3 single (FPAC by FPAC) 3-7, 10-50 single (FPAC by memory) 9-9, 10-49, 10-74f, 10-163 Multiply, narrow 10-97 narrow sign extend 2-5 signed 2-5, 10-91 unsigned 2-5, 10-90f wide 2-5, 10-141 wide signed 2-5, 10-141	and skip 2-3	
digit with overpunch 2-21, 10-29 float 10-28 floating-point 10-50 instruction, character 5-12 instructions, byte 2-20f decimal 2-20f fixed-point 2-3 single (FPAC by memory) 9-9, 10-49, 10-74f, 10-163 Multiply, narrow 10-97 narrow sign extend 2-5 signed 2-5, 10-91 unsigned 2-5, 10-90f wide 2-5, 10-141 wide signed 2-5, 10-141		
float 10-28 floating-point 10-50 instruction, character 5-12 instructions, byte 2-20f decimal 2-20f fixed-point 2-3 Multiply, narrow 10-97 narrow sign extend 2-5 signed 2-5, 10-91 unsigned 2-5, 10-90f wide 2-5, 10-141 wide signed 2-5, 10-141		
floating-point 10-50 instruction, character 5-12 instructions, byte 2-20f decimal 2-20f fixed-point 2-3 narrow 10-97 narrow sign extend 2-5 signed 2-5, 10-91 unsigned 2-5, 10-90f wide 2-5, 10-141 wide signed 2-5, 10-141		
instruction, character 5-12 instructions, byte 2-20f decimal 2-20f fixed-point 2-3 narrow sign extend 2-5 signed 2-5, 10-91 unsigned 2-5, 10-90f wide 2-5, 10-141 wide signed 2-5, 10-141		
instructions, signed 2-5, 10-91 byte 2-20f unsigned 2-5, 10-90f decimal 2-20f wide 2-5, 10-141 fixed-point 2-3 wide signed 2-5, 10-141		
byte 2-20f unsigned 2-5, 10-90f decimal 2-20f wide 2-5, 10-141 fixed-point 2-3 wide signed 2-5, 10-141	instruction, character 5-12	
decimal 2-20f wide 2-5, 10-141 fixed-point 2-3 wide signed 2-5, 10-141		
fixed-point 2-3 wide signed 2-5, 10-141		
		wide signed 2-5, 10-141
floating-point 3-3f	floating-point 3-3f	

N	overflow fault 5-21f
	parameters 9-2
N	pointer 5-21ff, 9-2
flag (negative flag) 3-8, 3-11	underflow 5-22
variable in PC relative addressing 1-11	underflow fault 5-21, 9-2
NADD 2-4, 10-91	stack,
NADDI 2-4, 10-91	defined 5-20
NADI 2-4, 10-92	defining limits of 9-2
Narrow	initializing 5-22
add 2-4, 10-91	location of 5-22
accumulator to memory word 2-4	pushing and popping 5-21
immediate 2-4, 10-77, 10-92, 10-166	store accumulator 2-3, 9-8, 10-80, 10-169
memory word to accumulator 2-4, 10-77, 10-166	subtract 2-4, 10-99
backward search queue and skip 5-3, 6-5, 10-92f	subtract
data, computing 2-2	immediate 10-79, 10-98, 10-168
decrement and skip if zero 2-9, 9-8, 10-78, 10-167	memory word 10-80, 10-169
divide 10-93	NBStc 5-3f, 6-5, 10-92f
divide memory word 2-5, 10-78, 10-167	NCLID 8-11, 10-93
DO until greater than 10-78, 10-167	NDIV 2-5, 10-93
extended add immediate 2-4, 10-91	NEG 2-6, 2-13f, 10-93ff
fault return block 5-19	Negate 2-13, 3-3, 10-50, 10-93ff
floating-point	Negate,
fault return block 5-16	narrow 2-13, 10-97
fault, servicing 5-15	optional shift logical 2-14
forward search queue and skip 5-3, 6-5, 10-95f	wide 2-13, 10-142
frame pointer 9-2	with optional skip 2-6, 2-13f
halve $(AC/2)$ 2-5, 10-61	Negative number, shifting a 2-6
immediate,	Never, skip 5-3
wide add with 10-142	Next instruction, addressing 5-1
wide arithmetic shift with 10-119	NFStc 5-3f, 6-5, 10-95f
wide logical shift with 10-138	NIO 7-3, 10-96, B-2
increment and skip if zero 2-9, 9-8, 10-79, 10-168	NLDAI 2-3, 10-96
load	NMUL 2-5, 10-97
accumulator 2-3, 9-8, 10-79, 10-168	NNEG 2-13, 10-97
CPU identification 8-11, 10-93	No
immediate 2-3, 10-96	error, skip on $(ANY = 0) 3-9, 10-57$
multiply 10-97	I/O transfer 7-3, 10-96
multiply memory word 2-5, 10-79, 10-168	mantissa overflow, skip on (MOF = 0) 3-9, 10-57
negate 2-13, 10-97	overflow and no zero divide, skip on (OVF and
queue, searching 6-5	DVZ = 0) 3-9, 10-58
return address 5-22	overflow,
block 9-2	skip on $(OVF = 0)$ 3-9, 10-57
block for decimal data, type 1 fault 5-19	skip on no underflow and (UNF and OVF = 0) 3-9,
block, pushing 5-15f, 5-22	10-58
sign extend divide or multiply 2-5	skip 5-3, 10-51
skip on	underflow and no overflow, skip on (UNF and
all bits set in accumulator 2-8, 10-97	OVF = 0) 3-9, 10-58
all bits set in memory 2-8, 10-97f	underflow and no zero divide, skip on (UNF and
any bit set in accumulator 2-8, 10-98	DVZ = 0) 3-9, 10-58
any bit set in memory 2-8, 10-98	underflow, skip on (UNF = 0) 3-9, 10-58
stack 1-4, 4-1, 5-15, 9-2, 9-4	zero divide,
fault 5-20	skip on (DVZ = 0) 3-9, $10-57$
fault handler 5-22f	skip on no overflow and (OVF and DVZ = 0) 3-9,
fault operations 5-21	10-58
fault return block 5-23	skip on no underflow and (UNF and DVZ = 0) 3-9,
fault, servicing 5-20, 5-22	Noninterruptible instructions 7.6
limit 5-21ff, 9-2	Noninterruptible instructions 7-6 Nonprivileged fault 1-16f, 5-13, 8-13f
management 1-4	pointer 5-13
operations, managing 4-1	servicing 5-14
- · · · · · · · · · · · · · · · · · · ·	or tions o it

Nonresident	wide
page 1-16	set bit to 9-8, 10-121f
page fault 5-13	skip on accumulator bit set to 2-23, 10-150
Nonsign-positioned numbers, for unpacked decimal 2-19f	skip on zero bit and set bit to 9-8, 10-156
Nonzero bit,	One's
	complement with optional skip 2-6, 2-14
skip on 9-5ff	
wide skip on 2-8, 2-14	complement, optional shift logical 2-14
Nonzero,	One-element queue 6-3
decrement and jump if 10-21	One-level page table 8-2f, 8-6ff
skip on $(Z = 0)$ 3-9, 10-57	Opcode pointer if
Normal	sign flag is zero 2-21
program execution 5-14f	trigger is one 2-21
program flow 5-1	Opcode,
Normalize 10-50	invalid 5-17, E-2
	I/O instruction 7-4
Normalize, floating-point 3-3	Opcodes,
Normalized format F-3	
Normalizing 3-4	interpreting LEF and I/O 7-1ff
floating-point data 3-1	wide instruction B-1
intermediate mantissa 3-5ff	Operand 1-12
the result 3-5	Operand,
Notation, maintaining excess 64 3-7f	bit 1-15
NSALA 2-8, 10-97	byte 1-13
NSALM 2-8, 10-97f	double word 1-2, 1-13
NSANA 2-8, 10-98	memory 1-12
· · · · · · · · · · · · · · · · · · ·	size 1-8
NSANM 2-8, 10-98	
NSBI 2-4, 10-98	word 1-2, 1-13
NSUB 2-4, 10-99	Operands, signs of the two 3-6
Number	Operating system, kernel 1-6
combination, for unpacked decimal sign and 2-19f	Operation,
of arguments 5-12f	extended 4-5, 4-8, 9-10f, 10-169f
of words required beyond WSL 4-8	finishing an I/O 7-5
too large to convert E-2	illegal I/O 8-3
Number,	initiating I/O 7-5
interrupting device 7-12f	I/O instruction 7-4
	legal I/O 8-3
physical page 7-2	
Numbers, compare two floating-point (set N and Z) 3-9	memory
Numeric with zero suppression, move 2-21, 10-30	read 8-5
Numerical algorithms, floating-point B-2	write 8-5
Numerics,	overflow stack 5-21, E-2
move 10-29	read 8-11
move j 2-21	stack 1-16
•	stack fault 5-13
0	testing the results of an 2-7
	underflow stack 5-21
Offset,	wide
page 8-7	
page table 8-3	extended 5-5, 9-10f, 10-102, 10-158
program counter 5-10ff	pop 5-20
termination 5-3	stack example 5-7f
word 1-15, 9-5f	write 8-11
	Operations,
Once, insert character 2-21, 10-25	binary 2-1
One to the intermediate mantissa, adding 3-6	byte 2-15
One,	checking for valid 5-13
set	decimal
bit to 9-5ff, 10-10	and byte 2-15
carry to 10-18	
T to 10-34	arithmetic 2-15
skip on zero bit and set bit to 9-5f, 9-8, 10-115f	device independent 7-3
, , , , , , , , , , , , , , , , , , ,	fixed-point logical 2-12
	floating-point arithmetic 3-4
	logical 2-1

managing	flag,
narrow stack 4-1	exponent 3-10
wide stack 4-1	mantissa 3-11
narrow stack fault 5-21	setting 2-8
shift 2-7	setting fixed-point 5-14
stack 4-1	mask,
wide	fixed-point 2-10, 9-10f
stack 4-1	wide save and reset 4-5, 5-5, 10-147
stack fault 5-20	wide save and set 4-5, 5-5, 10-147f
Option, skip 2-7	wide special save and reset 4-5, 5-5, 9-3, 9-10f,
Optional device flag handling 7-3	10-153
OR immediate,	wide special save and set 4-5, 5-5, 9-3, 9-10f, 10-153f
exclusive 2-13, 10-170	stack operation 5-21, E-2
inclusive 2-13, 10-64	Overflow,
wide	checking for
exclusive 2-13, 10-159	stack 5-14, 8-13
inclusive 2-12f, 10-135	vector stack 7-13
OR referenced bits 8-11, 10-99	exponent 3-6
OR,	fixed-point 2-6
exclusive 2-12f, 10-170	ignoring fixed-point 5-14
inclusive 2-12f, 10-63	mantissa 3-5f
wide	stack 4-6f, 8-13
exclusive 2-13, 10-159	Overpunch, move digit with 2-21, 10-29
inclusive 2-13, 10-135	Overwrite a page 8-1
Order of arguments 5-12	Overwriting data in area beyond the stack 5-20
ORFB 8-11, 10-99	OVF flag 3-6, 3-10, 5-15
Original	OVK
descriptor 5-18f	fixed-point overflow mask 9-10f
fault 8-13	mask 2-10, 3-11, 4-8, 5-14, 5-20, 7-13, 8-13
source indicator 5-18f	mask,
Outward	initializing 5-15, 7-13
call fault code E-1	resetting 2-9
return 5-8	setting 2-9, 5-20, 8-13
subroutine call, illegal 5-9	wide save/reset 4-8
Overflow	wide save/set 4-8
condition,	wide special save/reset 4-8
fixed-point 2-10	wide special save/set 4-8
stack 4-3, 5-12	to one,
detection, enabling vector stack 7-9	wide save and set 2-9f
fault 2-8	wide special save and set 2-9
mask, setting fixed-point 5-14	to zero,
service mask 1-2	wide save and set 2-9
fault,	wide special save and set 2-9
checking for stack 4-3	flag 2-8, 2-10, 5-14, 5-20, 7-13, 8-13
detecting an 2-10	flag,
detecting floating-point 5-15	initializing 5-15, 7-13
detecting wide stack 5-20	setting 5-20, 8-13
disabling narrow stack 5-22	skip on reset 2-8, 10-110
disabling wide stack 4-7, 5-20	Skip on reset 2 o, to tro
fixed-point 1-2, 2-8, 5-14f	P
ignoring floating-point 5-15	
initiating fixed-point 5-14	P depending on
narrow stack 5-21f	S, add to 10-20
servicing an 2-10	T, add to 10-21
faults, floating-point 5-15	Р,
flag 2-8, 2-10, 5-20, 7-13	add to 10-21
11ag 2-0, 2-10, 3-20, 1-13	current value of 5-18f

Packed	disk
byte 2-15	resident 8-4
decimal 2-15	resident memory 8-1
data, converting 2-20	invalid 8-4
string 2-17	location 8-1
Page 8-7, F-1	memory resident 8-4f, 8-11
access 8-4, 8-7	nonresident 1-16
access flag, valid 8-4	overwrite a 8-1
access,	physical 8-11
type of 8-4	referenced 8-1
valid 8-4	restricted 8-7
address, physical 8-6	valid 8-4, 8-7
	valid referenced 8-7
fault 8-4, 8-7, 8-10	Pages,
fault handler 5-13, 8-10f	defined 1-7
fault,	frequency of references to 8-11
cause of 8-10	status of 8-4
nonresident 5-13	
servicing 8-4, 8-10	swapping memory 8-7
from disk, loading referenced 8-10	Paging,
from memory, restoring 8-10	defined demand 8-10
level	mapping and demand F-1
one 8-7	Parameter, vector stack
two 8-7	fault address 7-9
number, physical 7-2	limit 7-9
offset 8-7	pointer 7-9
protocol access, ignoring 8-6f	Parameters of I/O data transfer 7-2
protocols 1-8f	Parameters,
table F-1	managing stack 4-1
address, physical 8-3	narrow stack 9-2
depth fault 8-6	pushing previously saved stack 7-9
entries 8-1	saving wide stack 7-9
entry 8-6f	storing processor 8-11
entry, format 8-4	vector stack 7-9
in memory 8-1	wide stack 4-1
logical word address, one-level or two-level 8-6	Passing
offset 8-3	arguments 5-12
	arguments to a subroutine 5-7
page, addressing another 8-6	PATU 8-11, 10-100
translation, example of one-level 8-8	PBX 2-9, 4-5, 4-8, 5-5f, 10-100
translation, example of two-level 8-9	PC 5-15f, 5-18f, 5-21, 5-23, 8-13, 9-4, E-2
translation, one-level 8-7	
translation, two-level 8-6f	and jump,
validity protection fault 8-7	pop 9-10f, 10-102
validity violation 8-13f	wide pop 4-5, 4-8, 5-5, 9-10f, 10-144
table,	relative
accessing 8-4	address mode 1-12
address of first entry in 8-2ff	addressing, n variable in 1-11
one-level 8-2f, 8-6ff	skip 5-3
two-level 8-2f	return address 4-6
tables 8-1, 8-4	Performing floating-point computations 2-15
zero 4-2, 5-10f, 5-13, 5-20, 5-22f, 8-10, 8-13f	Physical
Page,	address 1-7, 1-11, 7-2, 8-1ff
accessing physical 8-2	instruction, load 7-2
addressing another page table 8-6	into accumulator, loading 7-3
defined 8-1	to device, sending 7-2
	translation 8-3

address,	saving wide
forming 7-2	frame 7-9
logical address to 8-1	stack 7-9, 7-11
and conditional skip, load 10-81	setting
memory 8-1, 8-7	bit zero of narrow stack 5-22f
page 8-11	narrow stack 5-22
address 8-6	wide stack 4-3
number 7-2	single
table address 8-3	word 9-5f
page, accessing 8-2	word indirect 9-7
PIO 7-3, 10-101	skip on valid
Point	byte 10-116
location, binary 3-2	word 10-116f
	storing wide
to a segment, entry 5-9f	
Pointer 5-15, 7-6, 8-10, 8-13	frame 5-7, 5-11, 5-13, 8-10
chain 7-13	stack 5-11, 5-13, 8-10
contents, store accumulator in stack 10-111	updating wide stack 5-20
field, bit 1-15	using the wide frame 4-3
format, byte 1-14	wide
if sign flag is zero, opcode 2-21	frame 4-2f, 5-6ff, 7-9
if trigger is one, opcode 2-21	modify stack 4-4, 5-21, 9-11, 10-141
parameter, vector stack 7-9	stack 4-2f, 4-7, 5-12, 5-20, 7-9
register, initializing wide stack 8-13f	word 1-15, 5-12
to fault subopcode, byte 5-19	zero-extending vector stack 7-9
to subopcode causing fault, byte 5-18	Pointers,
to wide stack fault handler, saving 7-9	C/350 9-3
to word pointer, converting byte 2-21	trojan horse 5-12
wraparound, narrow stack 5-22	POP 9-8, 9-11, 10-101
Pointer,	Pop
bit 1-15, 9-5ff	block and
zero of narrow stack 5-22	execute 2-9, 4-5, 4-8, 5-5f, 5-23, 9-10f, 10-100,
zero of wide stack 5-21	10-102
byte 1-13f, 5-12, 9-5f	execute, wide 2-9, 4-5, 4-8, 5-8, 5-14, 5-21, 9-10f,
comparing wide stack 4-3	10-143
converting byte pointer to word 2-21	block, wide 2-9, 4-5, 4-8, 5-5f, 5-8, 5-14, 5-21, 9-10f
decrementing wide stack 4-3f	10-143
defined 1-11	context block, wide 2-9, 8-7, 8-10f, 10-129
forming	floating-point
bit 9-5f	
word 9-5f	state 3-10, 9-9, 10-51f
frame 9-2	status and accumulators 10-51f
	multiple
incrementing wide stack 4-3f	accumulators 9-8, 9-11, 10-101, 10-142
indirect 1-11, 5-10, 5-13ff, 5-17, 5-20, 5-23, 8-13f, 9-7,	accumulators, wide 2-3, 4-1, 4-5, 4-8, 9-8, 9-11,
9-9ff	10-142
initializing wide	operation, wide 5-20
frame 4-3, 5-11	PC and
stack 4-3, 8-10, 8-13f	jump 9-10f, 10-102
stack fault 7-9	jump, wide 4-5, 4-8, 5-5, 9-10f, 10-144
loading wide	Pop, wide floating-point 3-4, 4-5, 4-8, 10-131f
frame 5-12f	POPB 5-23, 9-10f, 10-102
stack 5-11, 5-13	POPJ 9-10f, 10-102
location bit 9-5f	Popping
modify stack 9-11, 10-90	accumulators 4-1
narrow	narrow stack 5-21
frame 9-2	return block 4-1, 4-5, 5-13
stack 5-21ff, 9-2	wide return block 4-3, 5-4, 7-7
nonprivileged fault 5-13	with WPOP 4-5
privileged fault 5-13	with WPOPJ 4-5
protecting against trojan horse 5-12	Port select instruction 7-2
proceeding against trojun notice J-12	Port, I/O (see I/O channel)

Power	counter,
fail flag 7-4f	floating-point 3-11
failure, detecting 7-5	format 1-5, 9-2
up 2-10	incrementing 3-9, 5-1
voltage ranges, detecting proper 7-5	jump relative to 9-10
Powers of 2 table D-1	loading 5-13
	_
Precision conversion, fixed-point 2-2	restoring 5-4
Precision,	updating 5-14, 5-20, 5-23
double 3-1f, 3-6f	development, supporting C/350 9-2
fixed-point data 2-2	execution, normal 5-14f
single 3-1f, 3-4, 3-6f	flow 5-2, 5-4
Previously saved stack parameters, pushing 7-9	instructions, C/350 9-2, 9-10
Priorities on faults 8-12f	management 1-5, 5-1
Priority of	management instructions, C/350 9-10
handling faults 5-13	flow,
	altering normal 5-1
handling I/O interrupts 5-14	
protection violation faults 8-13f	normal 5-1
Privileged	relative addressing 1-11
access fault 5-12	Program,
fault 1-16f	executing interrupted 5-14
fault pointer 5-13	expanding ECLIPSE C/350 9-3
fault, detecting 5-13	restarting interrupted 8-10
instruction violation fault code E-1	Programmed
instructions 8-2, 8-11	I/O 1-5, 7-1, 10-101
violation 8-13f	I/O command, issuing 7-3
	Programming, C/350 9-1
Privileges destination segment, access 5-12	
Processing, interrupt 5-6, 5-8, 7-7, 7-9, 7-11	Programs, supporting 16-bit and 32-bit 4-1
Processor	Protecting against trojan horse pointer 5-12
identification instructions, central 8-11	Protection F-2
instruction, equivalent 32-bit 9-7	capabilities, system 1-1
parameters, storing 8-11	fault 2-22, 5-11, 8-6f
state, internal 7-9	code 8-7, 8-15
status	code, validity bit E-1
register 1-2, 2-9, 5-12f, 5-15, 5-18, 5-21, 7-6, 8-13,	fault,
9-1, 9-4	initiating a 5-8
register format 2-10	page table validity 8-7
register from ACO, store 10-110	segment validity 8-2
	mechanism,
register in ACO, load 10-82	
register, setting 5-15ff, 7-3	accessing 1-17
storage capacity, exceeding 1-2	changing 1-17
Processor,	hierarchical 1-17
defined 1-1	violation 1-16, 8-2ff, 8-12ff
restoring state of 8-10	fault codes E-1
saving current state of 8-7, 8-10	fault handler 8-13f
state of the 8-10	fault, servicing 8-13f
Producing an intermediate	faults 1-8f, 5-13, 8-12ff
exponent 3-8	return block 8-13f
mantissa 3-8	violation,
	detecting 7-2
Program	
compatibility, upward 4-1, 9-1	indirect addressing 1-11
control to another segment, transferring 5-8ff	Protocols, page and segment 1-8f
control,	PRTSEL 7-2
returning 5-5f, 5-13	PSH 9-8, 9-11, 10-102f
to another segment transferring 5-8f	PSHJ 9-10f, 10-103
transferring 1-8, 4-2, 4-4, 5-6, 5-10, 5-16, 7-13, 8-13	PSHR 9-10f, 10-103
counter 1-5, 2-7, 2-22, 5-1ff, 5-11f, 5-17, 5-23, 7-13,	
9-2	
offset 5-10ff	
wraparound B-1	

PSR 4-6, 5-6f, 5-15, 5-18, 5-21, 7-12, 8-13, 9-3	link 6-2
from AC0, store 2-9	management 1-5, 6-1
in AC0, load 2-9f	Queue,
manipulation instructions 2-9	building a 6-1
word 8-13	defined 1-5, 6-1
word in device control table 7-13	empty 6-3
PSR, storing 5-7	one-element 6-3
PTE 8-4	ready 6-1
Purging address translator 8-11, 10-100	searching 6-5
Push	scarching 0-3
accumulators, wide 2-3, 4-5, 4-8, 9-8, 10-144	R
address 2-21, 4-8, 10-81, 10-171	
and jump 4-8	Range of
and jump 4-6 and jump to subroutine 4-5, 4-8, 5-6	displacement 1-11
	mantissa 3-2
byte address 2-21 effective	the intermediate mantissa 3-5
	the logical address, checking 8-6
address 2-21	Range,
byte address 2-20f, 4-5f, 4-8, 10-81, 10-171	addressing 1-10, 8-6, 9-10
floating-point state 3-10, 9-9, 10-52f	valid addressing 8-3
floating-point status and accumulators 10-52f	Ranges,
jump 5-5, 9-10f, 10-82, 10-103, 10-171	address 1-10, 1-12, 8-6, 9-10
multiple accumulators 9-8, 9-11, 10-102f	
return address 5-6, 9-10f, 10-103	detecting proper power voltage 7-5
Push, wide floating-point 3-4, 4-5, 4-8, 10-132f	Rates, data transfer 7-1
Pushed, number of arguments 5-12	Read
Pushing	access 1-9
a byte address 4-5	flag 8-5
a double word 4-3	violation 8-13f
a word address 4-5	access, valid 8-5
accumulators 4-1	command to device map, issuing 7-3
arguments onto a wide stack 5-7	data, accessing page to 8-4
byte address 4-5	high word 10-54
double word onto vector stack 7-12	operation 8-11
fault return block 5-14, 8-13	operation, memory 8-5
narrow	violation fault code E-1
fault return block 5-19	Read, valid 8-7
return block 5-15f, 5-22	Reading high floating-point word 3-3
stack 5-21	Ready queue 6-1
	Recognition,
previously saved stack parameters 7-9	disabling I/O interrupt 7-5, 7-12
return block 4-1, 4-5, 5-13, 5-16, 9-3	enabling
wide return block 5-4, 5-6, 5-12, 5-14ff, 5-18, 5-20	fixed-point fault 2-10, 5-14
wide return block onto vector stack 7-9, 7-11	floating-point fault 3-9ff, 5-15
with LPSHJ 4-5	I/O interrupt 7-5, 7-12
with WPSH 4-5	stack fault 5-20
with XPSHJ 4-5	Redefining wide stack 5-10f, 5-13
word address 4-5	Redefining wide stack 5-101, 5-13 Redefining wide stack for segment zero 8-10, 8-13f
0	Reference,
Q	I/O memory 8-11
Queue	
and skip,	valid segment 8-6
narrow backward search 5-3, 6-5, 10-92f	Referenced
	and modified bits, table of 8-1
narrow forward search 5-3, 6-5, 10-95f	bits,
wide backward search 5-3, 6-5, 10-120f	load modified and 8-11, 10-77
wide forward search 5-3f, 6-5, 10-133f	OR 8-11, 10-99
data element, dequeue a 10-22	reset 8-11, 10-103f
descriptor 6-2	store modified and 8-11, 10-109
element 5-4	flag 8-11
instructions 6-5	page 8-1
instructions, search 5-4	page from disk, loading 8-10

page, valid 8-7	Required
Register	beyond
assignments and format, map 7-2	WSL for stack fault, words 4-8
contents, modifying stack 4-2	WSL, number of words 4-8
format, processor status 2-10	memory 4-1, 5-11, 5-13ff, 5-17, 7-9, 7-11, 8-7, 8-10
instructions, wide stack 4-4	8-13f, 9-2
Register,	location, updating WSP 5-20
bit zero of segment base 8-2	words 5-20
floating-point status 1-3, 3-4, 3-6, 3-9f	Reset
	lead bit,
initializing wide stack 4-2, 4-4	locate and 2-13, 10-82
limit 8-13f	wide locate and 2-13, 10-138
pointer 8-13f	overflow mask,
load floating-point status 9-9, 10-48	wide save and 4-5, 5-5, 10-147
loading segment base 8-2	wide special save and 4-5, 5-5, 9-3, 9-10f, 10-153
modifying segment base 8-2	
processor status 1-2, 2-9, 5-12f, 5-15, 5-18, 5-21, 7-6,	referenced bits 8-11, 10-103f
8-13, 9-1, 9-4	Reset,
segment base 5-11, 7-1, 8-1f, 8-6	I/O 2-10, 7-3
setting processor status 5-15ff, 7-3	skip on OVR flag 2-8, 10-110
store floating-point status 9-9, 10-59, 10-74	system 2-10
testing the floating-point status 3-10	Resetting
wide stack	OVK mask 2-9
base 8-13f	TE mask 3-10
limit 8-13f	Resident
Registers,	flag, memory 8-4
C/350 9-1	memory page, disk 8-1
defined wide stack management 1-4	page,
fixed-point 1-2	disk 8-4
initializing wide stack 7-9, 7-11	memory 8-4f, 8-11
load all segment base 10-82f	Responding to interrupt request 7-6
	Restarting interrupted
map 7-2	instruction 7-6
saving wide stack 7-9	program 8-10
specifying eight segment base 8-6	Restore 9-10f, 10-104
stack 4-1	Restore from an I/O interrupt, wide 4-5, 5-8, 7-7
Relative	Restore, wide 2-9, 4-8, 9-10f, 10-144f
address mode, AC and PC 1-12	Restoring
address, effective or indirect 1-11	accumulators 5-4
addressing 1-10f, 1-13	
addressing,	carry 5-4
C/350 accumulator 9-1	page from memory 8-10
n variable in PC 1-11	program counter 5-4
program 1-11	state of processor 8-10
skip, PC 5-3	Restricted page 8-7
to program counter, jump 9-10	Restricting vector stack to lower 128 Kbyte 7-9
Removing guard digit 3-5	Restrictions, memory access 1-6
Replacing SAVE/RTN with	Result for a condition, testing 3-8
WSSVR/WRTN 9-4	Result,
WSSVS/WRTN 9-4	calculating the 3-5
Representation, excess 64 3-2	intermediate 3-6
Representing decimal digit and sign 2-16	normalizing the 3-5
Request,	rounding
access 8-7	the 3-5
blocking an interrupt 7-5f	the intermediate 3-11
detecting an I/O interrupt 5-1	shifting an intermediate 2-6
ignoring an interrupt 7-5	signs of the intermediate 3-5
	storing the floating-point 3-6
interrupt 7-6	
I/O interrupt 2-10, 4-1	
responding to interrupt 7-6	
servicing I/O interrupt 7-5, 8-13f	
Requesting I/O data transfer 7-5	

truncating	Return,
intermediate 3-11	illegal inward 5-9
the 3-5	outward 5-8
Results of an operation, testing the 2-7	subroutine 5-4, 5-13
Results, shifting decimal 2-22	wide 2-9, 4-8, 5-5, 5-8, 9-3f, 9-10f, 10-145
Resume	Returning
flag, interrupt (see IRES flag)	from
memory reference instruction 8-4	breakpoint handler 4-5
Resuming interrupted instruction 7-6	fault handler 3-11, 4-1
Retrieving data from the wide stack 4-3	I/O interrupt 4-1, 5-6, 5-8
Return 9-4, 9-10f, 10-105	LCALL 5-8
address 5-2, 5-7, 5-13, 5-15f, 5-20	subroutine call 3-11, 4-1, 4-3, 4-5
address to original fault 8-13	XCALL 5-8
address,	program control 5-5f, 5-13
narrow 5-22	with WPOPB 4-5
PC 4-6	Revision level, microcode 8-11
push 5-6, 9-10f, 10-103	Revision, floating-point 3-11
storing 5-7, 7-7	Right,
block 4-4, 5-12, 5-15f, 5-20f, 5-23, 9-10f, E-2	DHXR double hex shift 2-22, 10-23f
block	HXR single hex shift 2-22, 10-61
fault code, invalid microinterrupt E-1	shifting one bit to the 2-7
for ASCII data, type 2 or 3 fault 5-18f	wide move 2-21, 10-140
for decimal data, type 1 fault narrow 5-19	Ring, defined 1-6
for decimal data, type 1 fault wide 5-18	RND 3-11
instructions, wide stack 4-5	RND flag 3-4f, 3-11
onto vector stack, pushing wide 7-9, 7-11	Round flag 3-4f, 3-11
type 5-17	Rounding 3-4
block,	an intermediate mantissa 3-5ff
C/350 9-2	double to single, floating-point 3-3, 10-53f
defined wide 5-6f	down an even mantissa 3-5
fault 8-13	the intermediate result 3-11
fixed-point fault 5-15	the mantissa 3-4
floating-point fault 5-16	the result 3-5
narrow 9-2	up an odd mantissa 3-5
floating-point fault 5-16	Routine dependent information, interrupt 7-12
stack fault 5-23	Routine,
popping 4-1, 4-5, 5-13	device interrupt 7-12f
popping wide 4-3, 5-4, 7-7	interrupt service 7-5
protection violation 8-13f	transferring to device interrupt 7-13
pushing 4-1, 4-5, 5-13, 5-16, 9-3	RRFB 8-11, 10-103f
fault 5-14, 8-13	RSTR 9-10f, 10-104
narrow 5-15f, 5-22	RTN 9-4, 9-10f, 10-105
narrow fault 5-19	, ,
wide 5-4, 5-6, 5-12, 5-14ff, 5-18, 5-20	S
size 5-16	C - 11 (D 1 10 20
stack fault 4-8, 5-21, 5-23, 8-13	S, add to P depending on 10-20
standard wide 4-6	SAVE 5-21, 9-4, 9-10f, 10-105f, E-2
wide 4-3, 5-7, 5-20	Save 9-4, 9-10f, 10-105f
floating-point fault 5-15	and reset overflow mask,
stack fault 5-21	wide 4-5, 5-5, 10-147
blocks,	wide special 4-5, 5-5, 9-3, 9-10f, 10-153
types of narrow fault 5-19	and set overflow mask,
types of wide fault 5-18	wide 4-5, 5-5, 10-147f
instruction,	wide special 4-5, 5-5, 9-3, 9-10f, 10-153f
executing 5-1	and set OVK to one,
wide 5-6f, 5-13, 9-3f	wide 2-9f
via wide save, wide 4-5	wide special 2-9
	and set OVK to zero,
	wide 2-9
	wide special 2-9

instruction,	validity
wide 4-3, 5-12	flag 8-2
wide special 5-5ff, 9-3f	protection fault 8-2
without arguments 9-10f, 10-106f	violation 8-13f
Save, wide return via wide 4-5	zero 5-8, 5-13, 7-6f, 7-9, 7-11, 8-2, 8-10, 8-13f
Save/reset OVK mask,	zero,
wide 4-8	crossing to 7-9, 7-11, 8-13f
	redefining wide stack for 8-10, 8-13f
wide special 4-8	Segment,
SAVE/RTN with	access privileges destination 5-12
WSSVR/WRTN, replacing 9-4	accessing a 8-2
WSSVS/WRTN, replacing 9-4	accessing destination 5-9f
Save/set OVK mask,	another 4-2, 4-4, 5-6
wide 4-8	current 1-8, 4-1, 5-1, 5-6f, 5-13, 5-20, 5-23, 7-9, 7-11
wide special 4-8	8-10, 8-13f, 9-2, 9-4ff
Saved stack parameters, pushing previously 7-9	
Saving	defined 1-6
current state of processor 8-7, 8-10	current 1-4
pointer to wide stack fault handler 7-9	destination 1-9
wide	memory 1-4
frame pointer 7-9	destination 5-6f, 5-10ff
stack parameters 7-9	entry point to a 5-9f
stack pointer 7-9, 7-11	first 64 Kbytes of 9-2
stack registers 7-9	initiating a transfer to another 5-8
SAVZ 9-10f, 10-106f	invalid 8-2
SBI 2-4, 10-107	lower-numbered 4-3
SBR, setting bit two or three of 7-1ff	source 5-9ff
Scaling floating-point number 3-3, 10-54	transferring program control to another 5-8ff
Scan until	valid 8-3
true instruction, wide character 2-23	Segmentation F-1
true, wide character 10-126f	Segments, testing for identical 5-12
Scheme, addressing F-1	Select instruction, port 7-2
Search	Selecting LEF mode 7-1ff
queue and skip, narrow	Sending physical address to device 7-2
backward 5-3, 6-5, 10-92f	Sequence of subroutine instructions 5-6
forward 5-3, 6-5, 10-95f	Sequence,
queue and skip, wide	calling 5-13
backward 5-3, 6-5, 10-120f	DO-loop 5-3
forward 5-3f, 6-5, 10-133f	interrupt 7-8
queue instructions 5-4	Service
Searches a string of bytes 2-23	mask,
Searching queue 6-5	fault 1-3
Segment 4-1, 5-10, 8-4, 8-13, 9-2, 9-4ff, 9-10f	overflow fault 1-2
access 8-2	routine, interrupt 7-5
access,	Service, concluding vector interrupt 7-11
type of 8-2	Servicing
valid 8-2	a fault 1-2, 2-10, 3-11
base	a floating-point fault 3-9
register 5-11, 7-1, 8-1f, 8-6	an interrupt 7-6
registers, load all 10-82f	an interrupt request 7-5
call, too many arguments 5-21, E-2	an overflow fault 2-10
check, valid 8-3	base level vector interrupt 7-9, 10-144
crossing 5-6, 5-8, 5-21, 8-13f, E-2	fault 5-20
crossing violation 8-13f	floating-point fault 3-11, 5-15
	intermediate level vector interrupt 7-11ff, 10-143
field 1-8, 1-14f	I/O interrupt request 8-13f
field, modifying current 1-5	narrow
number 5-11	floating-point fault 5-15
of execution, changing current 7-6	stack fault 5-20, 5-22
protocols 1-8f	nonprivileged fault 5-14
reference, valid 8-6	original fault 8-13
transfer instructions 5-8	Oliginal lault 0-13

page fault 8-4, 8-10	floating-point fault
protection violation fault 8-13f	flags 5-15
vector interrupt 7-3	mask 5-15
wide	ION flag 7-6
floating-point fault 5-15	IRES flag 5-20, 7-6, 8-13
stack fault 5-20	modified flag 8-11
Set	N flag 3-8, 3-11
bit to one 9-5ff, 10-10	narrow stack
bit to one,	limit 5-22
skip on zero bit and 9-5f, 9-8, 10-115f	pointer 5-22
wide 9-8, 10-121f	overflow flag 2-8
wide skip on zero bit and 9-8, 10-156	
bit to zero 9-5ff, 10-10f	OVK mask 2-9, 5-20, 8-13
bit to zero, wide 9-8, 10-122	OVR flag 5-20, 8-13
	processor status register 5-15ff, 7-3
CARRY flag to one 2-6	referenced flag 8-11
CARRY flag to zero 2-6	RND flag 3-11
carry to one 10-18	TE mask 3-9ff, 5-15f
carry to zero 10-18	wide stack pointer 4-3
in accumulator,	WSP equal to WSL 5-20
narrow skip on all bits 10-97	Z flag (true zero flag) 3-8, 3-11
narrow skip on any bit 10-98	SEX 2-2, 10-108
wide skip on all bits 10-145	SGE 2-8, 10-108
wide skip on any bit 10-146	SGT 2-8, 10-108
in memory location,	Shift
narrow skip on all bits 10-97f	immediate, wide logical 10-139
narrow skip on any bit 10-98	instructions 2-6
wide skip on all bits 10-146	instructions,
wide skip on any bit 10-146	decimal 2-22
overflow mask,	hex 2-22
wide save and 4-5, 5-5, 10-147f	logical 2-14
wide special save and 4-5, 5-5, 9-3, 9-10f, 10-153f	wide arithmetic 2-6
sign flag to one or zero 2-21, 10-33	left immediate, wide logical 2-14
T to	left,
one 10-34	DHXL double hex 2-22, 10-23
zero 10-34	HXL single hex 2-22, 10-61
to one,	operations 2-7
bit 9-5ff	right,
wide skip on accumulator bit 2-23, 10-150	DHXR double hex 2-22, 10-23f
to zero,	HXR single hex 2-22, 10-61
bit 9-5ff	with narrow immediate,
wide skip on accumulator bit 2-23, 10-150	wide arithmetic 10-119
trigger to	wide logical 10-138
one 2-21	Shift,
zero 2-21	double logical 2-14, 10-27
Setting	logical 2-14, 10-84
ACO with address of fault instruction 8-13	wide
AC1 with value identifying fault 8-13	arithmetic 10-118f
an error flag 3-6	logical 2-14, 10-138
bit two or three of SBR 7-1ff	Shifting
bit zero of	0 to 31 bits 2-6
narrow stack limit 5-22f	a negative number 2-6
narrow stack pointer 5-22f	an intermediate mantissa 3-5
the WSL 5-20	an intermediate result 2-6
the WSP 5-20	decimal results 2-22
BUSY and DONE flag 7-5	one bit to the left 2-6
DVZ flag 3-8	one bit to the right 2-7
fixed-point overflow	the intermediate mantissa 3-6
fault mask 5-14	SI, add to 10-21
flag 5-14	-1, and to 10 21

Sign	load floating-point 9-9, 10-47, 10-72, 10-162
and number combination, for unpacked decimal 2-19f	multiply (FPAC by FPAC) 3-7, 10-50
bit 3-2	multiply (FPAC by memory) 9-9, 10-49, 10-74f,
bit change 2-6	10-163
bit, defined 2-2	store floating-point 9-9, 10-59, 10-75, 10-164
code, invalid 5-17, E-2	subtract (FPAC from FPAC) 3-7, 10-58
extend	subtract (memory from FPAC) 3-7, 9-9, 10-56, 10-74,
16-bits to 32-bits 2-2, 10-108	10-164
and divide 2-5, 10-27	Size field 2-16
divide, narrow 2-5	Size, memory 8-11
multiply, narrow 2-5	Skip
flag	always 10-54
is zero, opcode pointer if 2-21	and store if equal, wide mask 10-139f
to one or zero, set 2-21, 10-33	Skip
flag, insert character depending on 2-21	if accumulator
magnitude data 3-1	equal to immediate, wide 10-149
Sign,	greater than immediate, wide 2-8, 10-150
complementing the mantissa 3-7	greater than immediate, wide unsigned 2-8, 10-156
insert 10-25	less than or equal to immediate, wide 2-8, 10-151
load 9-8, 10-84	less than or equal to immediate, wide unsigned 2-8,
representing decimal digit and 2-16	10-157
wide load 2-13, 2-23, 9-8, 10-139	not equal to immediate, wide 2-8, 10-152
Sign-extending to 31 bits, displacement 1-11	if ACS
Signed	equal to ACD, wide 2-8, 10-148
divide 2-5, 10-26	greater than ACD 2-8, 10-108
divide, wide 2-5, 10-129	greater than ACD, wide signed 2-8,
divided instructions, C/350 B-2	10-149
multiply 2-5, 10-91	greater than ACD, wide unsigned 2-8,
multiply, wide 2-5, 10-141	10-157
skip if ACS	greater than or equal to ACD 2-8, 10-108
greater than ACD, wide 2-8, 10-149	greater than or equal to ACD, wide signed 2-8,
greater than or equal to ACD, wide 2-8, 10-149	10-149
less than ACD, wide 2-8	greater than or equal to ACD, wide unsigned 2-8,
less than or equal to ACD, wide 2-8, 10-151	10-157
less than, wide 10-151	less than ACD, wide signed 2-8
Significant	less than or equal to ACD, wide signed 2-8, 10-151
bit,	not equal to ACD, wide 2-8, 10-152
least 2-6f	if equal to immediate, wide 2-8 if less than, wide signed 10-151
most 2-7	
digit,	if zero 4-5 if zero,
least 2-17	decrement and 9-8, 10-34
most 2-17	extended decrement and 9-8, 10-36
Signs of the	extended increment and 9-8, 10-37
intermediate result 3-5	increment and 9-8, 10-64
two operands 3-6	narrow decrement and 2-9, 9-8, 10-78, 10-167
Single	narrow increment and 2-9, 9-8, 10-79, 10-168
hex shift	wide decrement and 2-9, 10-86, 10-174
left, HXL 2-22, 10-61	wide decrement and 2-9, 10-87, 10-174
right, HXR 2-22, 10-61	instruction,
precision 3-1f, 3-4, 3-6f	accumulator 5-2
word	device flags for 7-5
indirect pointer 9-7	executing 5-1
pointer 9-5f	instructions 2-7, 2-14, 2-22f, 3-8, 5-2f
Single, add (FPAC to FPAC) 3-6, 10-43	instructions, fixed-point logical 2-14
add (FPAC to FPAC) 3-6, 10-43 add (memory to FPAC) 3-6, 9-9, 10-42f, 10-71, 10-161	on accumulator bit
divide (FPAC by FPAC) 3-8, 10-421, 10-71, 10-101	instructions, wide 2-23
divide (FPAC by FFAC) 3-8, 10-43 divide (FPAC by memory) 3-8, 9-9, 10-44, 10-72,	set to one, wide 2-8, 2-23, 10-150
10-162	set to zero, wide 2-8, 2-23, 10-150
floating-point rounding double to 10-53f	

on all bits set in accumulator,	AND with optional 2-6, 2-12ff, 10-6ff
narrow 2-8, 10-97	compare to limits and 9-8, 10-12
wide 2-8, 10-145	fixed-point
on all bits set in memory,	decrement word and 2-9
narrow 2-8	increment word and 2-9
wide 2-8	increment with optional 2-4, 2-6, 2-8f
on any bit set in accumulator,	I/O 7-3, 10-109
narrow 2-8, 10-98	load physical and conditional 10-81
wide 2-8, 10-146	move
on any bit set in memory,	and 2-3
narrow 2-8	with optional 2-6, 2-8, 10-88ff
wide 2-8	narrow
on bit instruction, wide 2-14	backward search queue and 5-3, 6-5, 10-92f
on CARRY (see ADC, ADD, AND, COM, INC,	forward search queue and 5-3, 6-5, 10-95f
MOV, NEG, or SUB instruction)	negate with optional 2-6, 2-13f
on condition instructions,	never 5-3
floating-point 3-9	no 5-3, 10-51
logical 2-14	one's complement with optional 2-6, 2-14
on condition,	PC relative 5-3
fixed-point 2-7f	subtract with optional 2-4, 2-6, 2-8, 10-113ff
floating-point 3-9	wide .
testing BUSY or DONE flag and 7-3	backward search queue and 5-3, 6-5, 10-120f
on greater than or equal to zero $(N = 0)$ 3-9, 10-55	compare to limits and 2-8, 9-8, 10-122
on greater than zero (N and $Z = 0$) 3-9, 10-55	forward search queue and 5-3f, 6-5, 10-133f
on less than or equal to zero (N and $Z = 1$) 3-9, 10-55	Skipping
on less than zero $(N = 1) 3-9, 10-56$	a word 2-7
on no	an instruction 3-8
error $(ANY = 0)$ 3-9, 10-57	Skipping, word 2-7
mantissa overflow (MOF = 0) 3-9, 10-57	SKP 7-3, 10-109
overflow (OVF = 0) 3-9, $10-57$	SMRF 8-11, 10-109
overflow and no zero divide (OVF and DVZ = 0)	SNB 9-5ff, 10-109
3-9, 10-58	SNOVR 2-8, 10-110
underflow (UNF = 0) 3-9, $10-58$	Source
underflow and no overflow (UNF and OVF = 0)	indicator 2-21
3-9, 10-58	indicator, original 5-18f
underflow and no zero divide (UNF and DVZ = 0)	segment 5-9ff
3-9, 10-58	stack, copying arguments from the 5-12
zero divide (DVZ = 0) 3-9, 10-57	Source, AND with complemented 10-6
on nonzero $(Z = 0)$ 3-9, 10-57	Space,
bit 9-5ff, 10-109	C/350 address 9-4ff logical address 8-1, 8-10
bit, wide 2-8, 2-14, 9-8, 10-152	memory address 1-1
on OVR flag reset 2-8, 10-110	physical memory 8-10
on valid	valid address 2-22
byte 2-22	Special
pointer 10-116 word address 2-22	save and
	reset overflow mask, wide 4-5, 5-5, 9-3, 9-10f, 10-153
word pointer 10-116f	set overflow mask, wide 4-5, 5-5, 9-3, 9-10f, 10-153
on zero $(Z = 1)$ 3-9, 9-5ff, 10-55, 10-115 on zero bit 9-8	set OVK to one, wide 2-9
on zero bit and set bit to	set OVK to one, wide 2-9 set OVK to zero, wide 2-9
	save instruction, wide 5-5ff, 9-3f
one 9-5f, 9-8, 10-115f	save/reset overflow mask, wide 4-8
one, wide 2-8, 2-14, 9-8, 10-156	save/set overflow mask, wide 4-8
on zero bit and set to one, bit 9-8	Specific instructions, interrupting 2-11
on zero bit, wide 2-8, 2-14, 9-8, 10-156	Specifying eight segment base registers 8-6
option 2-7	SPSR 2-9, 10-110
Skip, add	STA 9-8, 10-110
complement with optional 2-4, 2-6, 2-8, 10-2f	0111 / 0, 10 110
with optional 2-4, 2-6, 2-8, 10-3ff	
always 5-3	

Stack 4-1	management
access instructions, double-word 4-5	instructions, C/350 9-11
base,	registers, defined wide 1-4
initializing wide 4-2, 8-10, 8-13f	management,
loading wide 5-11, 5-13	narrow 1-4
wide 4-2f, 4-7, 5-11, 5-13, 5-20, 7-9, 7-11, 8-10	wide 1-4
data instructions, wide 4-4	operation 1-16
example operation, wide 5-7f	operation example, wide 5-7
fault 4-1, 5-12, 5-14	operation,
address parameter, vector 7-9	overflow 5-21, E-2
code, wide 5-21, E-1f	underflow 5-21
handler,	operations 4-1
first instruction of vector 7-13	operations,
jumping to narrow 5-23	managing narrow 4-1
narrow 5-22f	managing wide 4-1
saving pointer to wide 7-9	wide 4-1
vector 7-13	overflow 4-6f, 8-13
wide 5-20	condition 4-3, 5-12
writing narrow 5-23	detection, enabling vector 7-9
writing wide 5-21	fault,
operation 5-13	checking for 4-3
operations,	checking for wide 4-3
narrow 5-21	detecting wide 5-20
wide 5-20	disabling narrow 5-22
pointer, initializing wide 7-9	disabling wide 4-7, 5-20
recognition, enabling 5-20	narrow 5-21f
return block 4-8, 5-21, 5-23, 8-13	overflow,
return block,	checking for 5-14, 8-13
narrow 5-23	checking for vector 7-13
wide 5-21	destination 5-12
fault,	parameters,
disabling 4-7, 5-20, 5-22	managing 4-1
narrow 5-20	narrow 9-2
servicing narrow 5-20, 5-22	pushing previously saved 7-9
servicing wide 5-20	saving wide 7-9
wide 5-20	vector 7-9
words required beyond WSL for 4-8	wide 4-1
faults,	pointer
type of 5-20	contents, store accumulator in 10-111
wide 4-7	parameter, vector 7-9
for segment zero, redefining wide 8-10, 8-13f	register, initializing wide 8-13f
for temporary storage, wide 5-16	wraparound, narrow 5-22
instructions,	pointer,
C/350 9-11	bit zero of
multi-word wide 4-8	narrow 5-22 wide 5-21
limit parameter, vector 7-9	comparing wide 4-3
limit,	decrementing wide 4-3f
bit zero of narrow 5-22	incrementing wide 4-3f
bit zero of wide 5-21	initializing wide 4-3, 8-10, 8-13f
initializing wide 8-10, 8-13f	loading wide 5-11, 5-13
loading wide 5-11, 5-13	modify 9-11, 10-90
lower 4-2	narrow 5-21ff, 9-2
narrow 5-21ff, 9-2	saving wide 7-9, 7-11
setting bit zero of narrow 5-22f	setting
setting narrow 5-22	bit zero of narrow 5-22f
upper 4-2 wide 4-3, 4-7, 5-11, 5-13, 5-20, 7-9, 7-11, 8-10	narrow 5-22
zero-extending vector 7-9	wide 4-3
location, defining 4-2	storing wide 5-11, 5-13, 8-10
iovation, defining 7 2	updating wide 5-20
	. ~

wide 4-2f, 4-7, 5-12, 5-20, 7-9	Starting
wide modify 4-4, 5-21, 9-11, 10-141	a device 7-5
zero-extending vector 7-9	address of block to transfer 7-2
register	address of older to transfer 7-2
contents, modifying 4-2	address of sub-block to transfer 7-2
instructions, wide 4-4	STASB 4-4, 10-111
register, initializing wide 4-2, 4-4	STASE 4-4, 10-111
	STASP 4-4, 10-111
registers 4-1f	State of processor,
registers,	restoring 8-10
initializing four wide 7-9, 7-11	saving current 8-7, 8-10
saving wide 7-9	State,
wide 4-2	device 7-5
return block instructions, wide 4-5	internal processor 7-9
to lower 128 Kbyte, restricting vector 7-9	pop floating-point 3-10, 9-9, 10-51f
underflow 4-7, 5-20, E-2	push floating-point 3-10, 9-9, 10-52f
condition 4-3	STATS 2-3, 4-5, 10-111
detection, enabling vector 7-9	Status
fault,	and accumulators,
detecting wide 5-20	
disabling narrow 5-22	pop floating-point 10-51f
disabling wide 4-7, 5-20	push floating-point 10-52f
narrow 5-21	field in context block 8-10
	flag, error 3-10
underflow,	flags, testing 3-8
enabling narrow 9-2	instructions, floating-point 3-10
narrow 5-22	of pages 8-4
wide 4-3	register
Stack,	format, processor 2-10
accessing double word in a 4-4	from ACO, store processor 10-110
C/350 9-2	in AC0, load processor 10-82
constructing double word for vector 7-12	register,
copying arguments from the source 5-12	floating-point 1-3, 3-4, 3-6, 3-9f
decrementing word in 2-21	load floating-point 9-9, 10-48
defined 1-4	processor 1-2, 2-9, 5-13, 5-15, 5-18, 5-21, 7-6, 8-13,
defined narrow 5-20	9-1, 9-4
defining limits of narrow 9-2	
initializing	setting processor 5-15ff, 7-3
narrow 5-22	store floating-point 9-9, 10-59, 10-74
wide 4-3, 4-6	testing the floating-point 3-10
loading the vector 7-9	Status,
location of narrow 5-22	floating-point 3-9
	load floating-point 10-73
lower limit of the wide 4-2	mantissa 3-9
narrow 1-4, 4-1, 5-15, 9-2, 9-4	storing floating-point 5-16
overwriting data in area beyond the 5-20	testing machine 5-2
popping narrow 5-21	STB 9-8, 10-112
pushing	STI 9-9, 10-112
arguments onto a wide 5-7	STIX 5-16f, 9-9, 10-113, E-2
word onto vector 7-12	STIX, destination indicator for 5-18f
narrow 5-21	Storage capacity, exceeding processor 1-2
wide return block onto vector 7-9, 7-11	Storage, wide stack for temporary 5-16
redefining wide 5-10f, 5-13	Store
retrieving data from the wide 4-3	
store in 2-21, 10-33	accumulator 9-8, 10-110
storing data in the 4-2	accumulator in
top location of wide 4-3	double word addressed by WSP 2-3, 4-5
upper limit of the wide 4-3	stack pointer contents 10-111
	WFP 4-4, 8-13f, 10-110
vector 7-7, 7-9, 7-12	WSB 4-4, 10-111
wide 1-4, 4-1, 5-4, 5-6f, 5-12f, 5-15	WSL 4-4, 10-111
Stack-referenced instructions, C/350 9-1	WSP 4-4, 10-111
STAFP 4-4, 10-110	
Standard wide return block 4-6	

accumulator,	Subprogram,
extended 9-8, 10-41	edit 5-4
narrow 2-3, 9-8, 10-80, 10-169	end edit 2-21
wide 2-3, 10-88, 10-175	WEDIT 5-4
byte 2-20, 9-8, 10-85, 10-112, 10-171	Subroutine 9-4
byte,	call 2-9, 3-11, 4-1, 4-3, 4-5, 4-8, 5-4ff, 5-8, 5-13
extended 9-8, 10-41	call,
wide 2-20, 9-8, 10-154	aborting 5-11f
floating-point	executing 5-1
double 9-9, 10-59, 10-75, 10-164	illegal outward 5-9
single 9-9, 10-59, 10-75, 10-164	returning from 3-11, 4-1, 4-3, 4-5
status register 9-9, 10-59, 10-74	from a 32-bit subroutine, calling C/350 9-4
if equal, wide mask skip and 10-139f	handler 2-10
in stack 2-21, 10-33	instructions, sequence of 5-6
integer 9-9, 10-112	return 5-4, 5-13
extended 9-9, 10-113	Subroutine,
extended, wide 9-9, 10-155	address first instruction of 5-10, 5-12
integer, wide 10-154f	call 10-65f, 10-159f
modified and referenced bits 8-11, 10-109	calling C/350 subroutine from a 32-bit 9-4
processor status register from AC0 10-110	changing C/350 9-4
PSR from AC0 2-9	executing 5-12 exiting 4-3
Storing	expanding an ECLIPSE C/350 9-4
accumulators 5-7	extended jump to 9-10, 10-37
byte address in accumulator 2-23	jump to 5-4ff, 9-3f, 9-10, 10-65, 10-76, 10-165
carry 5-7	jumping to 5-7
contents of WFP or WSP 8-13f	last instruction of 5-13
data in the stack 4-2	passing arguments to a 5-7
decimal number 2-20 device dependent constants or variables 7-12	push and jump to 4-5, 4-8, 5-6
floating-point data 3-4	returning from 3-11, 4-1, 4-3
floating-point status 5-16	Subtract
FPSR 3-10	double
processor parameters 8-11	(FPAC from FPAC) 3-7, 10-55, 10-58
PSR 5-7	(memory from FPAC) 3-7, 9-9, 10-56f, 10-74,
return address 5-7, 7-7	10-163
single precision 3-4	immediate 2-4, 10-107
the floating-point result 3-6	immediate,
transient overflow condition 2-10	narrow 10-79, 10-98, 10-168
wide frame pointer 5-7, 5-11, 5-13, 8-10	wide 10-88, 10-148, 10-175
wide stack pointer 5-11, 5-13, 8-10	instructions, decimal fixed-point 2-22
String	memory
of bytes,	word 2-4
manipulating 2-21	word, narrow 10-80, 10-169
moving 2-21	word, wide 10-88, 10-175f
searches a 2-23	single
of characters,	(FPAC from FPAC) 3-7, 10-58
converting 2-20	(memory from FPAC) 3-7, 9-9, 10-56, 10-74, 10-164
inserting 2-20	with optional skip 2-4, 2-6, 2-8, 10-113ff
String,	Subtract,
packed decimal 2-17	decimal 10-31f
unpacked decimal 2-16	narrow 2-4, 10-99
Structure, implicit data 2-15	wide 2-4, 10-155
SUB 2-4, 2-6, 2-8, 10-113ff	Subtraction instructions, fixed-point 2-4
Sub-block to transfer, starting address of 7-2	Subtraction, floating-point 3-5, 3-7
Subopcode causing fault, byte pointer to 5-18	Summary, instruction A-1ff
Subopcode, byte pointer to fault 5-19	Supporting
Subprogram instructions, EDIT 2-21	16-bit programs 4-1 32-bit programs 4-1
	C/350 program development 9-2
	devices 7-1
	ucvices /-1

Suppress, insert character 10-25	TE
Suppression, move numeric with zero 2-21, 10-30	fault mask 5-15
Swapping Swapping	flag 4-8
memory pages 8-7	mask,
two bytes 2-6	resetting 3-10
System	setting 3-9ff, 5-15f
	Temporary storage, wide stack for 5-16
calls, I/O 7-5	Termination offset 5-3
identification instructions 8-11	
management 1-6, 8-1	Testing
protection capabilities 1-1	a device flag 7-3f
reset 2-10	BUSY flag and skip on condition 7-3
System,	DONE flag and skip on condition 7-3
device interrupt 7-5	for a condition 2-23
interrupt 1-5, 7-5, 8-10	for identical segments 5-12
invoking interrupt 7-5, 8-10	interrupt level word 7-9
kernel operating 1-6	machine status 5-2
SZB 9-5ff, 10-115	result for a condition 3-8
SZBO 9-5ff, 10-115f	status flags 3-8
	the divisor for zero 3-8
T	the DVZ flag 3-9
— .	the floating-point status register 3-10
T to	the results of an operation 2-7
one, set 10-34	Three of SBR, setting bit 7-2
zero, set 10-34	Times, insert character j 10-25
T, add to P depending on 10-21	Top location of wide stack 4-3
Table	Towards the
address, physical page 8-3	head, enqueue 10-39f
depth fault, page 8-6	tail, enqueue 10-40
entries, page 8-1	Transfer
entry,	
page 8-6f	instructions, segment 5-8
vector 7-12	rates, data 7-1
in memory, page 8-1	to another segment, initiating a 5-8
logical word address,	Transfer,
one-level page 8-6	address of first word for I/O 7-2
two-level page 8-6	addressing memory for I/O data 7-2
of referenced and modified bits 8-1	direction
offset, page 8-3	of 5-8, 7-2
	of I/O data 7-2
page, addressing another page 8-6	initializing
translation,	burst multiplexor channel 7-2
example of one-level page 8-8	data channel 7-2
example of two-level page 8-9	no I/O 7-3, 10-96
one-level page 8-7	parameters of I/O data 7-2
two-level page 8-6f	requesting I/O data 7-5
validity	starting address of block to 7-2
protection fault, page 8-7	Transferring
violation, page 8-13f	blocks of words, between memory and device 1-6
Table,	bytes, between ac and device 1-5
accessing page 8-4	call arguments 5-10ff
address of first entry in page 8-2ff	data 1-5
base of vector 7-12	formatted data 2-20
device control 7-12f	
one-level page 8-2f, 8-6ff	I/O data 7-2
page F-1	page to disk 8-10
powers of 2 D-1	program control 1-8, 4-2, 4-4, 5-6, 5-10, 5-16, 7-13,
two-level page 8-2f	8-13
vector 7-11f	program control to another segment 5-8ff
word zero of vector 7-11	to device interrupt routine 7-13
	words, between
Tables, page 8-1, 8-4 Tail engueve towards the 10.40	ac and device 1-5
Tail, enqueue towards the 10-40	memory and device 1-6

Transfers between	Two's complement 2-2
device and accumulator 7-1	number,
device and memory 7-1	calculating a 2-8, 5-14
high speed device and memory 7-1	out of range calculating a 5-15
medium speed device and memory 7-1	Two-level page table 8-2f
memory and device 7-1ff	logical word address 8-6
Transfers, checking data 5-12	translation 8-6f
Transient overflow condition, storing 2-10	translation, example of 8-9
Translate and compare,	Type
character 9-8, 10-18f	6 or 7, invalid data 5-17, E-2
wide character 2-20, 2-23, 9-8, 10-127f	field 2-16
Translating logical	of data 2-16
	of page access 8-4
address 1-7, 7-2f, 8-2, 8-4	of segment access 8-2
address, most significant bits 7-2	of stack faults 5-20
Translation level flag 8-3	
Translation,	Type,
device address 7-1, 7-6	checking for valid data and 5-13, 5-16
example of	invalid sign code for data 5-17
one-level page table 8-8	return block 5-17
two-level page table 8-9	Types of
memory address 8-2f, 8-6	narrow fault return blocks 5-19
one-level page table 8-7	wide fault return blocks 5-18
two-level page table 8-6f	
Translator, purging address 8-11, 10-100	U
Trap	I IndonClass
disable,	Underflow
fixed-point 2-9, 10-60	condition, stack 4-3
floating-point 3-10, 10-60	detection, enabling vector stack 7-9
enable mask 3-9ff, 5-15	fault,
enable,	detecting floating-point 5-15
,	detecting wide stack 5-20
fixed-point 2-9, 10-60	disabling narrow stack 5-22
floating-point 3-10, 10-60	disabling wide stack 4-7, 5-20
Trap,	narrow stack 5-21
defined 5-14f	Underflow
floating-point B-2	faults, floating-point 5-15
Trigger	flag, exponent 3-10
is one, opcode pointer if 2-21	stack operation 5-21
to one or zero, set 2-21	Underflow,
Trigger, insert character depending on 2-21	enabling narrow stack 9-2
Trojan horse	exponent 3-6
pointer, protecting against 5-12	stack 4-7, 5-20, 5-22, E-2
pointers 5-12	wide stack 4-3
True	
value of exponent 3-2	UNF flag 3-10, 5-15
zero 3-1, 3-6, 3-11, F-2	Unique device, identifying 7-4
True, character move until 9-8, 10-13f	Unmapped
Truncating 3-4	memory addressing 1-5
intermediate	mode 7-3
mantissa 3-6ff	Unpacked
result 3-11	byte 2-15
	decimal 2-15
the mantissa 3-4	decimal string 2-16
the result 3-5	
Two	
bytes, swapping 2-6	
floating-point numbers, compare (set N and Z) 3-8f,	
10-43	
Kbyte address boundary 8-3	
of SBR, setting bit 7-1ff	
operands, signs of the 3-6	

Unsigned divide 2-5, 10-26 multiply 2-5, 10-90f	Value identifying fault, setting AC1 with 8-13
skip if	of exponent, true 3-2 of P, current 5-18f
accumulator greater than immediate, wide 2-8,	Value, absolute 10-42
10-156	Variable to constant, comparing DO-loop 5-3
accumulator less than or equal to immediate, wide 2-8, 10-157	Variable, incrementing a DO-loop 5-3 Variables, storing device dependent 7-12
ACS greater than ACD, wide 2-8, 10-157	VBP 2-22, 5-12, 10-116
ACS greater than or equal to ACD, wide 2-8, 10-157	Vector
Updating	interrupt
program counter 5-14, 5-20, 5-23	instruction 7-7
wide stack pointer 5-20	processing 7-7, 7-9
WSP reserved memory location 5-20	service, concluding 7-11
Upper	interrupt,
limit of the wide stack 4-3	I/O 2-9
stack limit 4-2	servicing 7-3
Upward program compatibility 4-1, 9-1	servicing base level 7-9, 10-144
Using	servicing intermediate level 7-11ff, 10-143
an accumulator 1-2	on interrupting device 10-172
device map 7-2	on I/O interrupt 4-8
the wide frame pointer 4-3	stack 7-7, 7-9, 7-12
V	stack fault
•	address parameter 7-9
Valid	fault handler 7-13
address space 2-22	limit parameter 7-9
addressing range 8-3	limit, zero-extending 7-9
ASCII data type 2-23	overflow checking for 7.13
byte pointer, skip on 10-116	overflow, checking for 7-13
byte, skip on 2-22	parameters 7-9
data and type, checking for 5-13, 5-16	pointer, zero-extending 7-9
decimal data type 2-23	to lower 128 Kbyte, restricting 7-9
gate 5-11	underflow detection, enabling 7-9 stack,
operations, checking for 5-13	constructing double word for 7-12
page 8-4, 8-7	loading the 7-9
page access 8-4	pushing double word onto 7-12
read 8-7	pushing wide return block onto 7-9, 7-11
read access 8-5	table 7-11f
referenced page 8-7	Vectored
segment 8-3	gate array 5-11
access 8-2	interrupt handler 7-7, 10-172
check 8-3 reference 8-6	Violation
	fault
word pointer, skip on 2-22, 10-116f write 8-7	codes E-1
Validate I/O access 8-2	codes, protection E-1
Validation, access 8-7	handler, protection 8-13f
Validity	fault,
bit protection fault code E-1	protection 1-8f, 5-13, 8-12ff
bit, checking 5-11	servicing protection 8-13f
flag,	return block, protection 8-13f
I/O 1-17, 7-2, 8-3	Violation,
segment 8-2	access (read, write, and execute) 8-13f
protection fault,	detecting protection 7-2
page table 8-7	indirect addressing 1-11, 8-13f
segment 8-2	inward reference 8-13f
violation, page table or segment 8-13f	I/O instruction 8-13f
	page table validity 8-13f
	privileged 8-13f

read access 8-13f	AND immediate 2-13, 10-118
segment	AND, with complemented source 2-13
crossing 8-13f	arithmetic shift 10-118f
validity 8-13f	instructions 2-6
write access 8-13f	with narrow immediate 10-119
Violations, multiple protection 8-12	backward search queue and skip 5-3, 6-5, 10-120f
Virtual	bit addressing format 1-15
memory 8-1	block move 2-3, 9-8, 10-119f
memory, defined 1-6	branch 9-10, 10-120
Voltage ranges, detecting proper power 7-5	character
	compare 2-23, 9-8, 10-123f
VWP 2-22, 5-12, 10-116f	move 2-20, 9-8, 10-125f
W	move until true 2-20, 9-8, 10-124f
••	scan until true 2-23, 10-126f
WADC 2-4, 10-117	translate and compare 2-20, 2-23, 9-8, 10-127f
WADD 2-4, 10-117	compare to limits and skip 2-8, 9-8, 10-127
WADDI 2-4, 10-117	
WADI 2-4, 10-117f	complement 10-126
WANC 2-13, 10-118	complement (one's complement) 2-13
WAND 2-13, 10-118	count bits 2-13, 9-8, 10-126
WANDI 2-13, 10-118	data, converting to 2-2
WASH 2-6, 10-118f	decrement and skip if zero 2-9, 10-86, 10-174
WASHI 2-6, 10-119	divide 2-5, 10-128
WBLM 2-3, 9-8, 10-119f	divide memory word 2-5, 10-85f, 10-173
WBR 5-2f, 9-10, 10-120	DO until greater than 10-86, 10-173f
WBStc 5-3f, 6-5, 10-120f	edit 4-8, 9-8, 10-129ff
WBTO 2-12f, 9-8, 10-121f	edit of alphanumeric 5-5
	exchange 2-3, 10-158
WBTZ 2-12f, 9-8, 10-122	exclusive
WCLM 2-8, 9-8, 10-122	OR 2-13, 10-159
WCMP 2-23, 9-8, 10-123f	OR immediate 2-13, 10-159
WCMT 2-20, 5-12, 9-8, 10-124f	extended operation 5-5, 9-10f, 10-102, 10-158
WCMV 2-20, 5-12, 9-8, 10-125f	fault return blocks, types of 5-18
WCOB 2-13, 9-8, 10-126	fix from floating-point accumulator 3-3f, 3-11, 9-9,
WCOM 2-13, 10-126	10-131
WCST 2-23, 10-126f	float from fixed-point accumulator 3-3f, 9-9, 10-13
WCTR 2-20, 2-23f, 9-8, 10-127f	floating-point
WDIV 2-5, 10-128	fault 5-15f
WDIVS 2-5, 10-129	pop 3-4, 4-5, 4-8, 10-131f
WDPOP 2-9, 8-10, 10-129	push 3-4, 4-5, 4-8, 10-132f
WEDIT 2-20ff, 4-8, 5-4f, 5-16f, 9-8, 10-129ff, E-2	forward search queue and skip 5-3f, 6-5, 10-133f
fault instruction, address of 5-18	frame pointer 4-2f, 5-6ff, 7-9
subprogram 5-4	frame pointer,
WFFAD 3-3f, 3-11, 9-9, 10-131	initializing 4-3, 5-11
WFLAD 3-3f, 9-9, 10-131	loading 5-12f
WFP 4-3, 4-6	saving 7-9
WFP,	storing 5-7, 5-11, 5-13, 8-10
load accumulator with 4-4, 10-67	using the 4-3
store accumulator in 4-4, 8-13f, 10-110	halve (AC/2) 2-5, 10-134
WFPOP 3-4, 3-10f, 4-5, 4-8, 9-9, 10-131f	immediate, wide
WFPSH 3-4, 3-10, 4-5, 4-8, 9-9, 10-132f	add with 10-117
WFStc 5-3f, 6-5, 10-133f	load with 10-135
WHLV 2-5, 10-134	inclusive
Wide	OR 2-13, 10-135
add 2-4, 10-117	
complement 2-4, 10-117	OR immediate 2-12f, 10-135 increment
immediate 2-4, 10-85, 10-117f, 10-173	
memory word to accumulator 2-4, 10-85, 10-172	(no skip) 2-4, 10-135
with narrow immediate 2-4, 10-142	and skip if zero 2-9, 10-87, 10-174
with wide immediate 2-4, 10-117	instruction opcodes B-1
AND 2-13, 10-118	instructions F-2
•	

load	signed skip if ACS
accumulator 2-3, 10-87, 10-174	greater than ACD 2-8, 10-149
byte 2-20, 9-8, 10-136	greater than or equal to ACD 2-8, 10-149
integer 9-9, 10-136	less than ACD 2-8
integer extended 9-9, 10-136f	less than or equal to ACD 2-8, 10-151
map 10-137	skip
sign 2-13, 2-23, 9-8, 10-139	if accumulator
with wide immediate 2-3, 10-135	equal to immediate 10-149
locate	greater than immediate 2-8, 10-150
and reset lead bit 2-13, 10-138	less than or equal to immediate 2-8, 10-151
lead bit 2-13, 10-137	not equal to immediate 2-8, 10-152
logical shift 2-14, 10-138	if ACS
immediate 2-14, 10-139	equal to ACD 2-8, 10-148
left immediate 2-14	not equal to ACD 2-8, 10-152
with narrow immediate 10-138	if equal to immediate 2-8
mask skip and store if equal 10-139f	on accumulator bit
modify stack pointer 4-4, 5-21, 9-11, 10-141	instructions 2-23
move 2-3, 10-140	set to one 2-8, 2-23, 10-150
move right 2-21, 10-140	set to zero 2-8, 2-23, 10-150
multiply 2-5, 10-141	on all bits set in
multiply memory word 2-5, 10-87, 10-175	accumulator 2-8, 10-145
• •	memory 2-8, 10-146
negate 2-13, 10-142	on any bit set in
pop block 2-9, 4-5, 4-8, 5-5f, 5-8, 5-14, 5-21, 9-10f,	accumulator 2-8, 10-146
	memory 2-8, 10-146
10-143	on bit instruction 2-14
block and execute 2-9, 4-5, 4-8, 5-8, 5-14, 5-21,	on nonzero bit 2-8, 2-14, 9-8, 10-152
9-10f, 10-143	
context block 2-9, 8-7, 8-10f, 10-129	on zero bit 2-8, 2-14, 9-8, 10-156 on zero bit and set bit to one 2-8, 2-14, 9-8, 10-156
multiple accumulators 2-3, 4-1, 4-5, 4-8, 9-8, 9-11,	
10-142	special save and reset overflow mask 4-5, 5-5, 9-3, 9-10f, 10-153
operation 5-20	set overflow mask 4-5, 5-5, 9-3, 9-10f, 10-153f
PC and jump 4-5, 4-8, 5-5, 9-10f, 10-144	
push accumulators 2-3, 4-5, 4-8, 9-8, 10-144	set OVK to one 2-9 set OVK to zero 2-9
restore 2-9, 4-8, 9-10f, 10-144f	special save instruction 5-5ff, 9-3f
restore from an I/O interrupt 4-5, 5-8, 7-7	
return 2-9, 4-8, 5-5, 5-8, 9-3f, 9-10f, 10-145	special save/reset OVK mask 4-8
block 4-3, 5-7, 5-20	special save/set OVK mask 4-8
block for decimal data, type 1 fault 5-18	stack 1-4, 4-1, 5-4, 5-6f, 5-12f, 5-15
block onto vector stack, pushing 7-9, 7-11	base 4-2f, 4-7, 5-11, 5-13, 5-20, 7-9, 7-11, 8-10
block, popping 4-3, 5-4, 7-7	base, initializing 4-2, 8-10, 8-13f
block, pushing 5-4, 5-6, 5-12, 5-14ff, 5-18, 5-20	base, loading 5-11, 5-13
block, standard 4-6	data instructions 4-4
instruction 5-6f, 5-13, 9-3f	example operation 5-7f
via wide save 4-5	fault 5-20
save and reset overflow mask 4-5, 5-5, 10-147	fault code 5-21, E-1f fault handler 5-20
save and set	fault handler, saving pointer to 7-9
overflow mask 4-5, 5-5, 10-147f	fault handler, saving pointer to 7-9 fault handler, writing 5-21
OVK to one 2-9f	fault nander, writing 3-21 fault operations 5-20
OVK to zero 2-9	
save instruction 4-3, 5-12	fault pointer, initializing 7-9
save, wide return via 4-5	fault return block 5-21
save/reset OVK mask 4-8	fault, servicing 5-20
save/set OVK mask 4-8	faults 4-7
set bit to	for segment zero, redefining 8-10, 8-13f
one 2-13, 9-8, 10-121f	for temporary storage 5-16 instructions, multi-word 4-8
zero 2-13, 9-8, 10-122	limit 4-3, 4-7, 5-11, 5-13, 5-20, 7-9, 7-11, 8-10
signed	limit, (also see WSL)
divide 2-5, 10-129	limit, (also see wSL) limit register, initializing 8-13f
multiply 2-5, 10-141	mint register, mittanizing 0-131

limit,	WINC 2-4, 10-135
bit zero of 5-21	WIOR 2-12f, 10-135
initializing 8-10, 8-13f	WIORI 2-12f, 10-135
loading 5-11, 5-13	WLDAI 2-3, 10-135
management 1-4	WLDB 2-20, 9-8, 10-136
operation example 5-7	WLDI 2-20, 3-3, 5-17, 9-9, 10-136, E-2
operations 4-1	WLDIX 2-20, 3-3, 5-17, 9-9, 10-136f, E-2
overflow fault,	WLMP 7-3, 10-137
	WLOB 2-13, 10-137
checking for 4-3	
detecting 5-20	WLRB 2-13, 10-138
disabling 4-7, 5-20	WLSH 2-14, 10-138
parameters 4-1	WLSHI 2-14, 10-138
parameters, saving 7-9	WLSI 2-14, 10-139
pointer 4-2f, 4-7, 5-12, 5-20, 7-9	WLSN 2-13, 2-23, 5-17, 9-8, 10-139, E-2
pointer, (also see WSP)	WMESS 10-139f
pointer register, initializing 8-13f	WMOV 2-3, 10-140
pointer,	WMOVR 2-21, 10-140
bit zero of 5-21	WMSP 4-4, 5-21, 9-11, 10-141, E-2
comparing 4-3	WMUL 2-5, 10-141
• •	WMULS 2-5, 10-141
decrementing 4-3f	
incrementing 4-3f	WNADI 2-4, 10-142
initializing 4-3, 8-10, 8-13f	WNEG 2-13, 10-142
loading 5-11, 5-13	Word
saving 7-9, 7-11	address field 1-14f
setting 4-3	address,
storing 5-11, 5-13, 8-10	one-level page table logical 8-6
updating 5-20	pushing 4-5
register instructions 4-4	skip on valid 2-22
register, initializing 4-2, 4-4	two-level page table logical 8-6
registers 4-2	addressed by WSP, store accumulator in double 4-5
registers,	addressing,
initializing 4-4, 7-9, 7-11	format C/350 9-5
saving 7-9	formats 1-9
	and skip, fixed-point
return block instructions 4-5	decrement 2-9
underflow 4-3	
underflow fault,	increment 2-9
detecting 5-20	block in a stack, accessing double 4-4
disabling 4-7, 5-20	boundary 1-2, 3-2
stack,	for I/O transfer, address of first 7-2
initializing 4-3, 4-6	for vector stack, constructing double 7-12
lower limit of the 4-2	in a stack, accessing double 4-4
pushing arguments onto a 5-7	in device control table, PSR 7-13
redefining 5-10f, 5-13	in stack, decrementing 2-21
retrieving data from the 4-3	indirect pointer, single 9-7
top location of 4-3	of interrupt handler, first 7-7
upper limit of the 4-3	offset 1-15, 9-5f
store	onto vector stack, pushing double 7-12
accumulator 2-3, 10-88, 10-175	operand 1-2, 1-13
byte 2-20, 9-8, 10-154	operand, double 1-2, 1-13
•	pointer 1-15, 5-12
integer 10-154f	
integer extended 9-9, 10-155	pointer,
subtract 2-4, 10-155	converting byte pointer to 2-21
immediate 10-88, 10-148, 10-175	forming 9-5f
memory word 10-88, 10-175f	single 9-5f
unsigned skip if	skip on valid 10-116f
accumulator greater than immediate 2-8, 10-156	to accumulator,
accumulator less than or equal to immediate 2-8,	narrow add memory 10-77, 10-166
10-157	wide add memory 10-85, 10-172
ACS greater than ACD 2-8, 10-157	zero of vector table 7-11
ACS greater than or equal to ACD 2-8, 10-157	

Word,	WSB,
decrementing interrupt level 7-9	load accumulator with 4-4, 10-67
incrementing interrupt level 7-9	store accumulator in 4-4, 10-111
load accumulator with double 10-67	WSBI 2-4, 10-148
loading current interrupt mask 7-12	WSEQ 2-8, 10-148
narrow	WSEQI 2-8, 10-149
divide memory 10-78, 10-167	WSGE 2-8, 10-149
multiply memory 10-79, 10-168	WSGT 2-8, 10-149
subtract memory 10-80, 10-169	WSGTI 2-8, 10-150
PSR 8-13	
	WSKBO 2-8, 2-23, 10-150
pushing a double 4-3	WSKBZ 2-8, 2-23, 10-150
read high 10-54	WSL 4-3
reading high floating-point 3-3	WSL for stack fault, words required beyond 4-8
testing interrupt level 7-9	WSL,
wide	load accumulator with 4-4, 10-67
divide memory 10-85f, 10-173	number of words required beyond 4-8
multiply memory 10-87, 10-175	setting
subtract memory 10-88, 10-175f	bit zero of the 5-20
Words required beyond	WSP equal to 5-20
WSL for stack fault 4-8	store accumulator in 4-4, 10-111
WSL, number of 4-8	WSLE 2-8, 10-151
Words, reserved memory 5-20	WSLEI 2-8, 10-151
WPOP 2-3, 4-1, 4-5, 4-8, 9-8, 9-11, 10-142	WSLT 2-8, 10-151
WPOP, popping with 4-5	WSNB 2-8, 2-14, 9-8, 10-152
WPOPB 2-9, 4-5, 4-8, 5-5f, 5-8, 5-14, 5-21, 9-10f, 10-143	WSNE 2-8, 10-152
WPOPB, returning with 4-5, 5-6	WSNEI 2-8, 10-152
WPOPJ 4-5, 4-8, 5-5f, 9-10f, 10-144	WSP 4-3
WPOPJ, popping with 4-5	equal to WSL, setting 5-20
WPSH 2-3, 4-1, 4-5, 4-8, 9-8, 9-11, 10-144	reserved memory location, updating 5-20
WPSH, pushing with 4-5	WSP,
Wraparound,	decrement double word addressed by 2-9, 4-5, 10-34
address 1-4f, 5-1, B-1	increment double word addressed by 2-9, 4-5, 10-64
narrow stack pointer 5-22	load accumulator
program counter B-1	with 4-4, 10-67
Write	with double word addressed by 4-5
access 1-9	modifying 4-4
flag 8-5	setting bit zero of the 5-20
violation 8-13f	store accumulator
command to device map, issuing 7-3	in 4-4, 10-111
data, accessing page to 8-4	in double word addressed by 2-3, 4-5
operation 8-11	storing contents of 8-13f
operation, memory 8-5	WSSVR 2-9, 4-3, 4-5, 4-8, 5-5f, 5-21, 9-3f, 9-10f, 10-153
violation fault code E-1	E-2
Write, valid 8-7	WSSVR/WRTN, replacing SAVE/RTN with 9-4
Writing	WSSVS 2-9f, 4-3, 4-5, 4-8, 5-5ff, 5-21, 9-3f, 9-10f,
narrow stack fault handler 5-23	10-153f, E-2
to memory 8-7	WSSVS/WRTN, replacing SAVE/RTN with 9-4
wide stack fault handler 5-21	WSTB 2-20, 9-8, 10-154
WRSTR 2-9, 4-5, 4-8, 5-8, 7-7, 9-10f, 10-144f	WSTI 2-20, 2-23, 3-3, 5-17, 9-9, 10-154f, E-2
WRTN 2-9, 4-4f, 4-8, 5-5ff, 5-13, 9-3f, 9-10f, 10-145	WSTI, destination indicator for 5-18f
WSALA 2-8, 10-145	WSTIX 2-20, 3-3, 5-16f, 9-9, 10-155, E-2
WSALM 2-8, 10-146	WSUB 2-4, 10-155
WSANA 2-8, 10-146	WSZB 2-8, 2-14, 9-8, 10-156
WSANM 2-8, 10-146	WSZBO 2-8, 2-14, 9-8, 10-156
WSAVR 2-9, 4-5, 4-8, 5-5ff, 10-147, E-2	WUGTI 2-8, 10-156
WSAVR, executing 5-13	WULEI 2-8, 10-157
WSAVS 2-9f, 4-5, 4-8, 5-5ff, 10-147f, E-2	WUSGE 2-8, 10-157
WSAVS, executing 5-13	WUSGT 2-8, 10-157
WSB 4-2	WXCH 2-3, 10-158
	WXOP 4-5, 4-8, 5-5f, 9-10f, 10-102, 10-158

X

X instruction prefix 1-11
XCALL 2-9, 4-4ff, 4-8, 5-4ff, 10-159f
XCALL count 4-6
XCH 2-3, 10-160
XCT 5-1f, 10-160f
XFAMD 3-6, 9-9, 10-161
XFAMS 3-6, 9-9, 10-161
XFDMD 3-8, 9-9, 10-161
XFDMS 3-8, 9-9, 10-162
XFLDD 3-4, 9-9, 10-162
XFLDS 3-4, 9-9, 10-162
XFMMD 3-7, 9-9, 10-163
XFMMS 3-7, 9-9, 10-163
XFSMD 3-7, 9-9, 10-163
XFSMS 3-7, 9-9, 10-164
XFSTD 3-4, 9-9, 10-164
XFSTS 3-4 9-9 10-164
XFSTS 3-4, 9-9, 10-164 XJMP 5-2, 7-7, 9-10, 10-165 XJSR 5-4ff, 9-3f, 9-10, 10-165
XISR 5-4ff 9-3f 9-10 10-165
XLDB 2-20, 9-8, 10-165
XLEF 2-12ff, 2-21, 9-10, 10-165
XLEFB 2-21, 10-166 XNADD 2-4-10-166
XNADD 2-4, 10-166 XNADI 2-4, 10-166
XNDIV 2-5, 10-167
XNDO 5-3, 10-167
XNDSZ 2-9, 9-8, 10-167
XNISZ 2-9, 9-8, 10-168
XNLDA 2-3, 9-8, 10-168 XNMUL 2-5, 10-168
XNSBI 2-4, 10-168
XNSTA 2-3, 9-8, 10-169
XNSUB 2-4, 10-169
XOP0 9-10f, 10-169f
XOR 2-12f, 10-170
XORI 2-13, 10-170
XPEF 2-21, 4-5, 4-8, 5-7, 10-171
XPEFB 2-21, 4-5f, 4-8, 10-171
XPSHJ 4-5, 4-8, 5-5f, 9-10f, 10-171
XPSHJ, pushing with 4-5
XSTB 2-20, 9-8, 10-171
XVCT 2-9, 4-8, 7-7, 9-10, 10-172
XVCT displacement 7-11
XWADD 2-4, 10-172
XWADI 2-4, 10-173
XWDIV 2-5, 10-173
XWDO 5-3, 10-173f
XWDSZ 2-9, 10-174
XWISZ 2-9, 10-174
XWLDA 2-3, 10-174
XWMUL 2-5, 10-175
XWSBI 2-4, 10-175
XWSTA 2-3, 10-175
XWSUB 2-4, 10-175f

Z flag, setting (true zero flag) 3-8, 3-11 Zero
bit and set
bit to one, skip on 9-5f, 9-8, 10-115f
bit to one, wide skip on 9-8, 10-156
to one, skip on 9-8
bit,
skip on 9-8
wide skip on 9-8, 10-156
extend 16-bits to 32-bits 2-2, 10-176
flag, mantissa divide by 3-11
suppression, move numeric with 10-30
Zero,
crossing to segment 7-9, 7-11, 8-13f
decrement and skip if 9-8, 10-34
divisor equals 3-11
extended decrement and skip if 10-36
impure F-2
increment and skip if 10-64
narrow decrement and skip if 9-8, 10-78, 10-167
narrow increment and skip if 9-8, 10-79, 10-168
page 4-2, 5-10f, 5-13, 5-20, 5-22f, 8-10, 8-13f
redefining wide stack for segment 8-10, 8-13f
segment 5-8, 5-13, 7-6f, 7-9, 7-11, 8-2, 8-10, 8-13f
set
bit to 9-5ff, 10-10f
carry to 10-18
T to 10-34
skip
if 4-5
on $(Z = 1)$ 3-9, 9-5ff, 10-55, 10-115
true 3-6, F-2
wide
decrement and skip if 10-86, 10-174
increment and skip if 10-87, 10-174
set bit to 9-8, 10-122
skip on accumulator bit set to 2-23, 10-150
Zero-extending vector stack limit and pointer 7-9
Zero-extends 1-11
ZEX 2-2, 10-176
LLA 2-2, 10-1/U

CUT ALONG DOTTED LINE

1-Data General USEPS GPOUP

Installation Membership Form

Name		Position _		Date
Company, Organization or School	il			
Address		City	Stat	e Zip
Telephone: Area Code	No		Ext	
1. Account Category	☐ OEM ☐ End User ☐ System F	House	5. Mode of Operation	□ Batch (Central) □ Batch (Via RJE) □ On-Line Interactive
2. Hardware M/600 C/350, C/330, C/300 S/250, S/230, S/200 S/130 AP/130 CS Series N3/D Other NOVA microNOVA Other (Specify)	Qty. Installed	Qty. On Order	7. Application Description	□ RSTCP □ CAM □ HASP □ 4025 □ RJE80 □ Other □ SAM Specify □ □
3. Software 4. Languages	☐ AOS ☐ DOS ☐ SOS Specify — ☐ Algol ☐ DG/L ☐ Cobol ☐ ECLIPSE Cobol	□ RDOS □ RTOS □ Other □ Assembler □ Interactive □ Fortran	8. Purchase 9. Users Group	From whom was your machine(spurchased? Data General Corp. Other Specify Are you interested in joining a special interest or regional Data General Users Group?
	☐ Business BASIC ☐ BASIC Specify	□ RPG II □ PL/1 □ Other		O

FOLD

TAPE

FOLD TAPE

FOLD FOLD



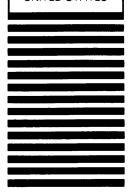
BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 26 SOUTHBORO, MA. 01772

Postage will be paid by addressee:



ATTN: Users Group Coordinator (C-228) 4400 Computer Drive Westboro, MA 01581 NO POSTAGE NECESSARY IF MAILED IN THE UNITED STATES





TTD.	
112	

TIPS ORDER FORM Technical Information & Publications Service

BILL T	`O:		SHIP TO: (if different)			
COMPA	ANY NAME	MECOMPANY NAME				
ADDRI	ESS		ADDRESS			
CITY_	····		CITY			
STATE	`	ZIP	STATE	ZIP		
ATTN:			ATTN:			
QTY	MODEL #	DE	SCRIPTION	UNIT	LINE	TOTAL
				PRICE	DISC	PRICE
-						
					110 90	
(Additi	ional items ca	an be included on second order f	form) [Minimum order is \$50.00]	ТОТ	AL	
			Tax Exempt # or Sales Tax (if applicable)	Sales Tax		
				Shipping		
				тот	AL	
		ETHOD OF PAYMENT — ney order enclosed	SHIP VI			
		ss than \$100.00	□ DGC will select best way (U.□ Other:	P.S or Po	stal)	
		☐ Visa ☐ MasterCard Expiration Date	☐ U.P.S. Blue Label			
	Purchase Ord	der Number:				
	N	OTE: ORDERS LESS THAN \$100,	INCLUDE \$5.00 FOR SHIPPING AND HAI	NDLING		
Person	to contact abo	out this order	Phone	Ex	tension	
Mail Ord	lers to:	-	Buyer's Authorized Signature			Deta
Attn: E	eneral Corporate Educational Services Computer Drive	tion vices/TIPS F019	(agrees to terms & conditions on reverse side)			Date
Westbo	oro, MA 01580 17) 366-8911 ex		Title			
		-	DGC Sales Representative (If Known)		Bad	ge#

DISCOUNTS APPLY TO MAIL ORDERS ONLY



DATA GENERAL CORPORATION TECHNICAL INFORMATION AND PUBLICATIONS SERVICE TERMS AND CONDITIONS

Data General Corporation ("DGC") provides its Technical Information and Publications Service (TIPS) solely in accordance with the following terms and conditions and more specifically to the Customer signing the Educational Services TIPS Order Form shown on the reverse hereof which is accepted by DGC.

1. PRICES

Prices for DGC publications will be as stated in the Educational Services Literature Catalog in effect at the time DGC accepts Buyer's order or as specified on an authorized DGC quotation in force at the time of receipt by DGC of the Order Form shown on the reverse hereof. Prices are exclusive of all excise, sales, use or similar taxes and, therefore are subject to an increase equal in amount to any tax DGC may be required to collect or pay on the sale, license or delivery of the materials provided hereunder.

2. PAYMENT

Terms are net cash on or prior to delivery except where satisfactory open account credit is established, in which case terms are net thirty (30) days from date of invoice.

3. SHIPMENT

Shipment will be made F.O.B. Point of Origin. DGC normally ships either by UPS or U.S. Mail or other appropriate method depending upon weight, unless Customer designates a specific method and/or carrier on the Order Form. In any case, DGC assumes no liability with regard to loss, damage or delay during shipment.

4 TFRM

Upon execution by Buyer and acceptance by DGC, this agreement shall continue to remain in effect until terminated by either party upon thirty (30) days prior written notice. It is the intent of the parties to leave this Agreement in effect so that all subsequent orders for DGC publications will be governed by the terms and conditions of this Agreement.

5. CUSTOMER CERTIFICATION

Customer hereby certifies that it is the owner or lessee of the DGC equipment and/or licensee/sub-licensee of the software which is the subject matter of the publication(s) ordered hereunder.

6. DATA AND PROPRIETARY RIGHTS

Portions of the publications and materials supplied under this Agreement are proprietary and will be so marked. Customer shall abide by such markings. DGC retains for itself exclusively all proprietary rights (including manufacturing rights) in and to all designs, engineering details and other data pertaining to the products described in such publication. Licensed software materials are provided pursuant to the terms and conditions of the Program License Agreement (PLA) between the Customer and DGC and such PLA is made a part of and incorporated into this Agreement by reference. A copyright notice on any data by itself does not constitute or evidence a publication or public disclosure.

7. DISCLAIMER OF WARRANTY

DGC MAKES NO WARRANTIES, EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, WARRANTIES OF MERCHANT-ABILITY AND FITNESS FOR PARTICULAR PURPOSE ON ANY OF THE PUBLICATIONS SUPPLIED HEREUNDER.

8. LIMITATIONS OF LIABILITY

IN NO EVENT SHALL DGC BE LIABLE FOR (I) ANY COSTS, DAMAGES OR EXPENSES ARISING OUT OF OR IN CONNECTION WITH ANY CLAIM BY ANY PERSON THAT USE OF THE PUBLICATION OF INFORMATION CONTAINED THEREIN INFRINGES ANY COPYRIGHT OR TRADE SECRET RIGHT OR (II) ANY INCIDENTIAL, SPECIAL, DIRECT OR CONSEQUENTIAL DAMAGES WHATSOEVER, INCLUDING BUT NOT LIMITED TO LOSS OF DATA, PROGRAMS OR LOST PROFITS.

9. GENERAL

A valid contract binding upon DGC will come into being only at the time of DGC's acceptance of the referenced Educational Services Order Form. Such contract is governed by the laws of the Commonwealth of Massachusetts. Such contract is not assignable. These terms and conditions constitute the entire agreement between the parties with respect to the subject matter hereof and supersedes all prior oral or written communications, agreements and understandings. These terms and conditions shall prevail notwithstanding any different, conflicting or additional terms and conditions which may appear on any order submitted by Customer.

DISCOUNT SCHEDULES

DISCOUNTS APPLY TO MAIL ORDERS ONLY.

LINE ITEM DISCOUNT

5-14 manuals of the same part number - 20% 15 or more manuals of the same part number - 30%



014-000704-02