Data General

Customer Documentation

16-bit Real-Time ECLIPSE®
Assembly Language Programming

16-Bit Real-Time ECLIPSE®
Assembly Language Programming

Data General

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.

CEO, DASHER, DATAPREP, ECLIPSE, ENTERPRISE, INFOS, microNOVA, NOVA, PROXI, SUPERNOVA, ECLIPSE MV/4000, ECLIPSE MV/6000, ECLIPSE MV/8000, TRENDVIEW, MANAP, SWAT, GENAP, and **PRESENT** are U.S. registered trademarks of Data General Corporation, and **AZ-TEXT, ECLIPSE MV/10000, DG/L, DG/XAP, GW/4000, GDC/1000, REV-UP, UNX/VS, XODIAC, DEFINE, SLATE, DESKTOP GENERATION, microECLIPSE, BusiPEN, BusiGEN,** and **BusiTEXT** are U.S. trademarks of Data General Corporation.

Ordering No. 014-000688

© Data General Corporation, 1984, 1986
All Rights Reserved
Printed in the United States of America
Rev. 02, May 1986

Preface

This manual provides extensive information for assembly language programmers working with 16-bit real-time ECLIPSE® computers. It explains the 16-bit processor-independent concepts, functions, and instruction set. The processor-dependent information can be found in a companion manual — a processor-specific assembly language programming manual.

Organization

16-Bit Real-Time ECLIPSE *Assembly Language Programming* contains eight chapters and one appendix.

Chapter 1, "System Overview," explains the system components and functions of interest to programmers of 16-bit real-time ECLIPSE computers.

Chapter 2, "Memory Access and Stack Management," explains memory referencing and stack operation and summarizes stack instructions.

Chapter 3, "Fixed-point Computation," gives fixed-point data formats, explains fixed-point operations, and summarizes the fixed-point instructions.

Chapter 4, "Floating-point Computation," gives floating-point data formats, explains floating-point operations, and summarizes the floating-point instructions.

Chapter 5, "Program Flow Management," explains program flow, and fault handling and summarizes the instructions that alter program flow.

Chapter 6, "Device Management," explains the ECLIPSE I/O interrupt system, summarizes the general I/O and interrupt system instructions, and discusses interrupts and interrupt handling.

Chapter 7, "Memory and System Management," describes the 16-bit ECLIPSE translation facilities and memory protection features and summarizes their instructions for the operating system designer. It also discusses the special system functions for the operating system designer.

Chapter 8, "Instruction Dictionary," explains each instruction. Dictionary entries are organized alphabetically by assembler mnemonic.

Appendix A, "Commercial Instructions," presents the instructions that differentiate the commercial ECLIPSE (C-series) computers from the real-time ECLIPSE (S-series) computers.

Conventions and Abbreviations

This manual uses the following conventions and abbreviations to present assembler statements for instructions.

UPPERCASE	Uppercase characters indicate a literal argument (command mnemonic) in an assembler statement. When you include a literal argument with an assembler statement, use the exact form.
variable	Lowercase 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.
[variable]	Italicized square brackets indicate an optional argument. Omit the square brackets when you include an optional argument with an assembler statement.
ac	The abbreviation ac indicates a fixed-point accumulator.
acs	The abbreviation acs indicates a source fixed-point accumulator.
acd	The abbreviation acd indicates a destination fixed-point accumulator.
fpac	The abbreviation fpac indicates a floating-point accumulator.
facs	The abbreviation facs indicates a source floating-point accumulator.
facd	The abbreviation facd indicates a destination floating-point accumulator.

In addition, diagrams in the following format show the arrangement of the 16 bits in a word or fixed-point accumulator.

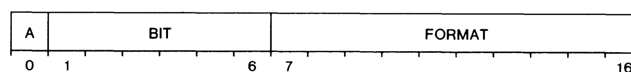


Table of Contents

Preface

Organization i
Conventions and Abbreviations i

Chapter 1
System Overview

Memory Access 1
Stack Management 1
Fixed-point Computation 3
Floating-point Computation 3
Program Flow Management 4
Device Management 4
Memory Management 4
System Management 4

Chapter 2
Memory Access and Stack Management

Memory Access Overview 5
Effective Addresses 6
- Addressing Modes 6
- Lower Page Zero 6
- Indirect Addresses 7

Operand Access 7
- Word/Double-Word Access 7
- Byte Access 7
- Bit Access 8

Memory Protection 9
Stack Management 9
Stack Operations 9
- Stack Base 9
- Stack Limit 9
- Stack Pointer 10
- Frame Pointer 10

Stack Parameter Instructions 10
Stack Data Instructions 10
Stack Initialization 11
Stack Protection 12
- Stack Overflow 12
- Stack Underflow 12

Chapter 3
Fixed-point Computation

Data Formats 13
- Arithmetic Data 13
- Logical Data 13
- Decimal Data 14
- Byte Data 14

Common Operations 14
- Carry Operations 14
- Shift Operations 14
- Skip Operations 15

Move Instructions 15
Arithmetic Instructions 15
Logical Instructions 17
Decimal/Byte Instructions 18

Chapter 4
Floating-point Computation

Floating-point Data 19
Floating-point Operations 19
- Guard Digits 20
- Mantissa Alignment 20
- Calculation and Normalization 20
- Truncation and Storage 20

Data Conversion Instructions 20
Move Instructions 20
Addition and Subtraction Instructions 21
- Addition 21
- Subtraction 21

Multiplication and Division Instructions 21
- Multiplication 21
- Division 22

Skip Instructions 22
Status and Faults 23
Status Instructions 24

Chapter 5
Program Flow Management

Execute Instruction (XCT) **25**
Jump Instructions **25**
Skip Instructions **25**
Subroutine Calls and Returns **26**
  Programming Examples **30**
Fault Handling **31**
  Stack Faults **32**
  Floating-point Faults **33**

Chapter 6
Device Management

Device Access **35**
General I/O Instructions **36**
Interrupt System **37**
Interrupt System Instructions **37**
Interrupts **37**
  Instruction Interruption **37**
  Interrupt Mask **38**
  Interrupt Level **38**
Interrupt Servicing **38**
Vector Interrupt Processing **39**

Chapter 7
Memory and System Management

Memory Allocation and Protection **41**
Address Translation **41**
Protection **42**
  Validity Protection **42**
  Write Protection **43**
  Indirection Protection **43**
  I/O Protection **43**
Protection Fault Handling **43**
Address Translator Status **44**
Address Translator Instructions **44**
Error Checking and Correction **45**
  ERCC Instructions **45**
System Status and Special Functions **45**

Chapter 8 47
Instruction Dictionary

Appendix A
Commercial Instruction Set

Data Formats **115**
Move and Sign Instructions **117**

Index **127**

DG Offices

How to Order Technical Publications

Technical Products Publications Comment Form

Users' Group Membership Form

Chapter 1

System Overview

The heart of a 16-bit real-time ECLIPSE computer is its processing system — hereafter called the *processor* — which accesses memory, manages data, and controls program flow.

The processor performs fixed-point computation in all ECLIPSE computers and floating-point computation in most ECLIPSE computers. It also manages stacks, program flow, peripheral devices, and memory. The processor performs these tasks using the facilities of the memory and I/O systems as shown in Figure 1.1. The memory system provides main memory and the facilities for managing main memory and maintaining its integrity. The I/O system provides facilities for servicing NOVA/ECLIPSE peripheral devices.

Memory Access

The processor uses a 15-bit address to access memory. The program can specify an address either *absolutely* by giving the actual address or *relatively* by giving a displacement relative to the contents of the program counter or an accumulator. In addition, the program can also use the address *indirectly* as a pointer to a memory location containing yet another address.

Chapter 2, "Memory Access and Stack Management," summarizes addressing concepts and provides information on memory reference instructions.

Stack Management

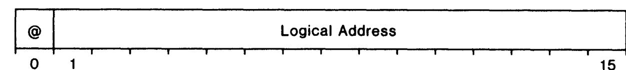
A *stack* is a series of consecutive locations in memory. Typically, a program uses a stack to pass arguments between subroutine calls and to save the program state when servicing a fault. After executing a subroutine or fault handler, the processor restores the program state and continues program execution.

The processor manages the stack with three 16-bit parameter registers:

- *Stack limit* which defines the upper limit of the stack;
- *Stack pointer* which addresses the current location on the stack;
- *Frame pointer* which defines a reference point within the stack.

The format of these registers is diagrammed and explained in the next section.

Stack Management Register Format



NOTES: *@ (bit 0) is the indirect address bit.*

The lowest numbered bit of a register (such as bit 0) is the most significant bit. The highest numbered bit (such as bit 15) is the least significant bit.

The processor accesses these stack management registers to save and restore stack parameters when changing program flow. The program accesses them with instructions that load or store a register value.

Chapter 2, "Memory Access and Stack Management," summarizes stack concepts and provides information on stack instructions.

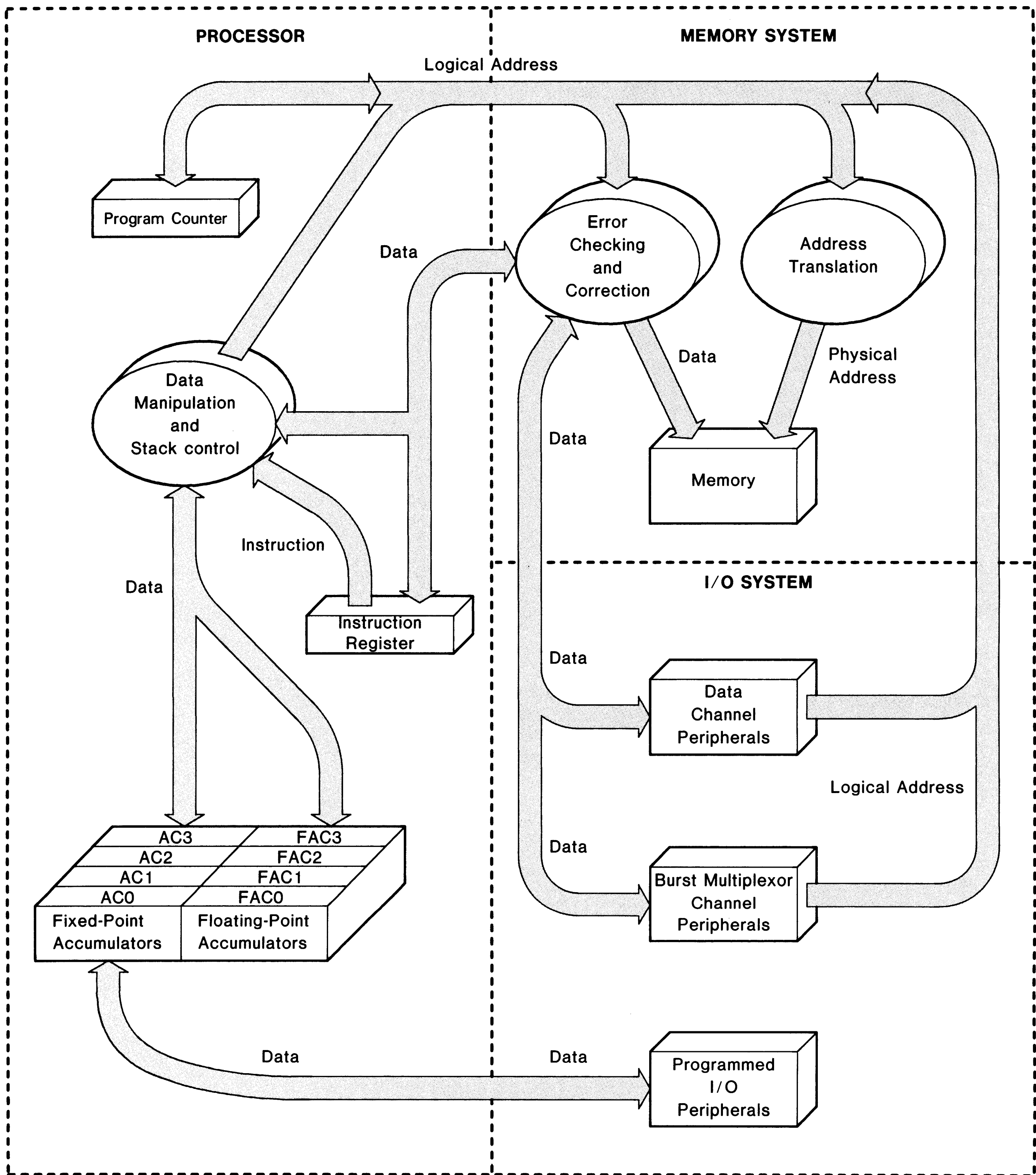


Figure 1.1 Functional components of a typical 16-bit real-time ECLIPSE computer system

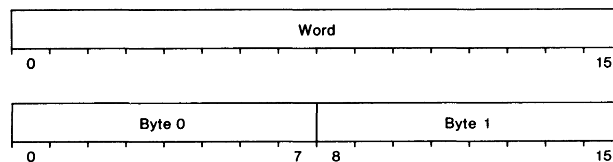
Fixed-point Computation

Fixed-point computation consists of

- Fixed-point binary arithmetic with signed and unsigned 16-bit and 32-bit numbers;
- Logical operations with 16-bit or 32-bit binary data;
- Decimal arithmetic with unsigned 4-bit packed decimal numbers;
- Character manipulation with 8-bit bytes.

To facilitate fixed-point computation, the processor contains four 16-bit fixed-point accumulators, AC0 through AC3. The format of fixed-point accumulators are diagrammed and explained in the next section.

Fixed-point Accumulator Formats



A program accesses a fixed-point accumulator with instructions that manipulate a bit, byte, word, or double word. A word or double word operand must begin on a word boundary (bit 0); a byte must begin on a byte boundary (bit 0 or bit 8).

Chapter 3, "Fixed-point Computation," provides additional information.

In addition to using an accumulator for fixed-point computation, the program can

- Use AC2 or AC3 in relative memory addressing in place of the program counter.
- Load or build an instruction in an accumulator and execute it.

Chapter 2, "Memory Access and Stack Management," provides additional information on addressing modes; Chapter 5, "Program Flow Management," provides additional information on instruction execution.

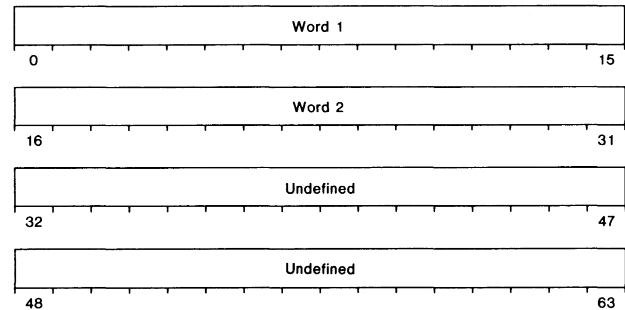
Floating-point Computation

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

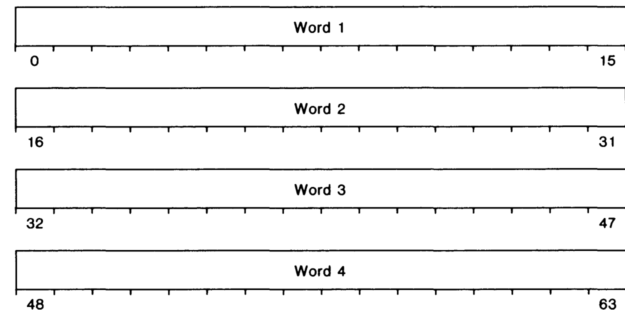
To facilitate floating-point computations, the processor contains four 64-bit floating accumulators, FPAC0 through FPAC3, and a floating-point status register, FPSR. The formats of floating-point accumulators are diagrammed below.

Floating-point Accumulator Formats

Single Precision



Double Precision



A program accesses a floating-point accumulator with instructions that manipulate single- and double-precision numbers. A single-precision number requires two consecutive words; a double-precision number requires four consecutive words.

Chapter 4, "Floating-point Computation," provides additional information.

The 32-bit floating-point status register contains the following flags:

overflow and underflow fault flags,
mantissa status flag,
divide-by-zero flag,
fault service (trap) mask,
processor status flags.

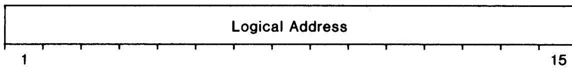
The processor sets an overflow fault, underflow fault, or mantissa status flag when the result of a floating-point computation exceeds the processor storage capacity. The divide-by-zero flag indicates an attempt to divide a number by zero. The fault service mask determines whether or not the processor can service a fault. The remaining flags provide processor status.

The program can access the contents of the register with instructions to initialize it or to test a flag and skip on a condition. Chapter 4, "Floating-point Computation," provides further information on the floating-point status register and the instructions that access it.

Program Flow Management

Program flow management consists of controlling program execution — calling a subroutine, for instance — and handling faults. The processor controls program flow with a 15-bit program counter (PC) which holds the logical address of the executing instruction. During normal program flow the processor increments the program counter. Thus address wraparound can occur. The format of the program counter is diagrammed below.

Program Counter Format



Device Management

Device management involves transferring data between memory and a device. The processor can transfer data — bytes, words, or blocks of words — with the programmed I/O (PIO) and data channel (DCH) facilities. Some processors can also transfer data with a high-speed burst multiplexor channel facility. Common to all three facilities are the I/O instructions, address translation, and the interrupt system.

Chapter 6, "Device Management," provides additional information on the interrupt facility and I/O instructions; Chapter 7, "Memory and System Management," provides additional information on address translation.

Using the *programmed I/O facility*, the program transfers bytes or words between a fixed-point accumulator and a device. The program uses the programmed I/O facility to transfer data with a slow speed device such as a terminal, or to set up a data channel or burst multiplexor channel transfer.

Using the *data channel facility*, the program sets up a transfer of words between memory and a device; the device initiates the transfer of each word. The data channel accesses memory directly (with or without address translation). Thus, the data transfer bypasses the accumulators.

Using the *burst multiplexor channel facility*, the program sets up a transfer of blocks of words between memory and a device; the device initiates the transfer of each block. Like the data channel, the burst multiplexor channel accesses memory directly (with or without address translation). Thus, the data transfer bypasses the accumulators.

Memory Management

Programs use logical addresses to access locations in physical memory. The processor supports a logical address space of 64 Kbytes and from 64 Kbytes to 2048 Kbytes of physical memory, depending on the memory system. Both logical and physical address spaces are organized into pages of 2 Kbytes.

In systems with more than 64 Kbytes of physical memory, physical memory is greater than logical memory. In these systems, the processor uses address translation to convert (*map*) a logical page address into a physical page address.

The hardware facilities for address translation include several map tables that store information indicating where logical pages reside in physical memory. Each *map table* contains thirty-two program-accessible registers (*entries*), one register (*entry*) for each logical page. The hardware facilities also contain a program-accessible address translation status register that selects a map table for initialization or translation and provides memory access protection to improve memory integrity. By manipulating the contents of these registers, the operating system software can allocate physical memory to various software functions and define memory restrictions.

In addition to the memory access protection, most memory systems also provide an error checking and correction facility to improve memory integrity. When enabled, this facility calculates and writes a check code to memory when data is written to memory. When the data is read from memory, the facility recalculates the check code and compares it to the original code to determine if the data was read correctly.

Chapter 7, "Memory and System Management," provides additional information on address translation and error checking and correction.

System Management

The processor provides several special functions for system management. Some functions are provided by all 16-bit real-time ECLIPSE processors, others are provided only by specific types of processors. Refer to the Chapter 7, "Memory and System Management," for additional information.

Chapter 2

Memory Access and Stack Management

Memory Access Overview

The processor addresses and accesses memory for an instruction or for an operand. The program specifies the address of the instruction or operand with a memory reference instruction.

The instructions that the processor accesses can be single- or double-word. The operand can be a bit, byte, word, a double word, or two double words. A 16-bit word is the standard unit of address for all memory access, regardless of the size of the unit.

Memory reference instructions are a class of instructions that access memory for another instruction or an operand. Each instruction contains information for determining the effective (logical) address of an operand or the next nonsequential instruction. After determining the effective address of an operand, the processor reads or writes the operand. In the case of the next nonsequential instruction, the processor loads the effective address into the program counter and then executes the instruction that the program counter identifies.

A memory reference instruction attempts to access memory in the user's 64 Kbyte logical address space. If user address translation is disabled, access to any address is valid. If user address translation is enabled, the validity of the access depends on the page protection features defined for the addressed page by the user's map table. These features can provide valid page protection, write protection, I/O access protection, and indirect protection. If these features are violated, the processor aborts the access and services the protection violation fault.

Refer to Chapter 7, Memory and System Management, for further details on memory reference protection.

Typical memory reference instruction formats are diagrammed below. These formats contain the following fields:

op code (OP),
index
displacement
optional indirect (@)
accumulator (AC or FPAC)

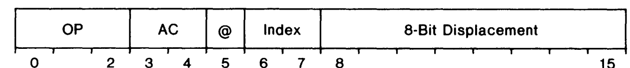
The op code field (OP) defines the operation that the instruction performs. Operations include load and store accumulator, add, subtract, multiply, divide two numbers, etc.

The combination of the index, displacement, and indirect fields specify parameters of the effective (logical) address for the instruction or operand. Except for instructions which address bits, all memory reference instructions use either an 8-bit (short) or a 15-bit (extended) displacement field. If the mnemonic for a memory reference instruction has an E or F prefix, the instruction uses a 15-bit displacement; all other memory reference instructions use the standard 8-bit displacement.

In the diagrams below, the accumulator field specifies a source or destination accumulator ranging from 0 to 3. For instance, when the ac field for a load accumulator instruction, is 0 (ac = 0) the processor loads an operand from memory into the destination fixed-point accumulator (AC0) for a fixed-point instruction or into the destination floating-point accumulator (FPAC0) for a floating-point instruction.

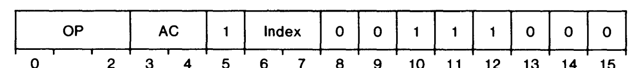
Typical Fixed-point Instruction Format (8-Bit Displacement)

Word 1



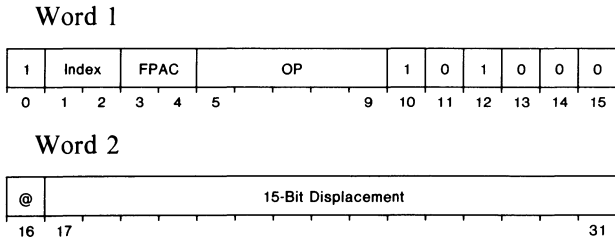
Typical Fixed-point Instruction Format (15-Bit Displacement)

Word 1



Word 2

Typical Floating-point Instruction Format (15-Bit Displacement)



Effective Addresses

To resolve the effective address, the processor first identifies the addressing mode, then any indirect address(es), and finally the effective address.

Addressing Modes

Using the index field defined in Table 2.1, the processor determines if the instruction specifies an absolute or relative addressing mode. The DGC assembler in conjunction with the appropriate pseudo-op produces object code with absolute or relative displacements.

For *absolute addressing*, the unmodified displacement field contains an indirect or an effective address. The address, expressed as an 8-bit or 15-bit unsigned integer, specifies an addressing range as shown in Table 2.1.

For *relative addressing*, the index field specifies either the program counter, AC2, or AC3, the contents of which become a base address. The processor adds the base address to the displacement, which is expressed as a signed 8-bit or 15-bit integer, to obtain an indirect or effective address. Before adding an 8-bit displacement, the processor sign extends it to 15 bits. When executing an instruction with a 15-bit displacement, the processor adds one to the sum for program-counter relative addressing. This additional increment adjusts the sum to address the displacement, which is in the word after the word containing the instruction op code. An instruction with an 8-bit displacement contains the displacement in the same word as the op code.

Lower Page Zero

Regardless of the addressing mode, any memory reference instruction can address the first 400_8 (256_{10}) locations in page 0 of the logical address space. For this reason, these locations have special significance: they are collectively referred to as *lower page zero.* Locations 0 through 47_8 are reserved for special purposes. Certain of these reserved locations serve the same purpose in all 16-bit processors. Table 2.2 lists these locations and their functions.

The functions of the other reserved locations vary with the processor. For more information on these locations, refer to the assembly language programming manual for the specific computer. The remaining locations in lower page zero comprise a common storage area for items used throughout a program.

Addressing Mode	Index Field	Intermediate Logical Address[1]	Displacement Length	Displacement Ranges (Words) Octal	Decimal
Absolute	0 0	Displacement	8 bits	0 to 377	0 to 255
		Displacement	15 bits	0 to 77777	0 to 32,767
PC Relative	0 1	PC +/− Displacement	8 bits	−200 to +177	−128 to +127
		PC + 1 +/− Displacement	15 bits	−40000 to +37777	−16,384 to +16,383
AC2 Relative	1 0	AC2 +/− Displacement	8 bits	−200 to +177	−128 to +127
		AC2 +/− Displacement	15 bits	−40000 to +37777	−16,384 to +16,383
AC3 Relative	1 1	AC3 +/− Displacement	8 bits	−200 to +177	−128 to +127
		AC3 +/− Displacement	15 bits	−40000 to +37777	−16,384 to 16,383

Table 2.1 Effective Addressing Ranges

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

Location (octal)	Name	Function
0	I/O return	Return address from I/O interrupt. Also address of first instruction of autorestart routine.
1	I/O handler	Address of the I/O interrupt handler.
2	System call handler	Address of the system call instruction handler.
3	Protection fault handler	Address of the protection fault handler.
4	Vector stack pointer	Address of the top of the vector stack.
5	Current mask	Current interrupt priority mask.
6	Vector stack limit	Address of the last normally useable location in the vector stack.
7	Vector stack fault handler	Address of the vector stack fault handler.
20-27	Autoincrement	In the earlier computers, the contents of any of these locations are automatically incremented by one when the location is indirectly referenced.
30-37	Autodecrement	In the earlier computers, the contents of any of these locations are automatically decremented by one when the location is indirectly referenced.
40	Stack pointer	Address of top of stack.
41	Frame pointer	Address of frame reference within the stack.
42	Stack limit	Address of the last normally usable location in the stack.
43	Stack fault handler	Address of the stack fault handler.
44	XOP table	Address of the beginning of the extended operation table.
45	Floating-point fault handler	Address of the floating-point fault handler.

Table 2.2 Standard functions of reserved memory locations in lower page zero.

Indirect Addresses

When the indirect field is 0, the absolute or relative address becomes the effective address. In systems with address translation, the user address translation facility converts the effective address (the logical address) into a physical address which the processor accesses. In systems without address translation, or with user address translation disabled, the effective address is the physical address.

When the indirect field is 1, the absolute or relative address becomes an indirect address or pointer. The user address translation facility converts the address to a physical address and uses the contents of that physical address as another indirect or direct (effective) address.

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

- When bit 0 is 0, the contents are the effective address, which is translated into physical address and accessed.
- When bit 0 is 1, the contents are another pointer. In this case, the processor continues to resolve pointers until it reaches one with bit 0 equal to 0.

Most processors can resolve only a limited number of pointers when address translation is enabled. An attempt to resolve more results in protection violation. For more information on indirect addressing for a particular processor, refer to the assembly language programming manual for that processor.

Operand Access

Before accessing an operand in memory, the processor first resolves the effective address. The processor accesses the operand as a bit, byte, word, or double word.

Word/Double-Word Access

The processor accesses a word operand for fixed-point computation. An instruction that requests a word, such as *Load Accumulator* (LDA), supplies the effective address parameters to the processor. The processor then resolves the effective address and accesses the addressed word.

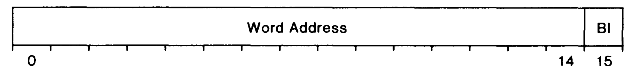
The processor accesses a double word operand for floating-point computation. A single precision floating-point instruction requires one double word and a double-precision instruction requires two double words. All floating-point mnemonics have an F prefix; those with an S suffix are single-precision instructions; those with a D suffix are double-precision instructions.

An instruction that requests a double word, such as *Load Floating-Point Single* (FLDS), 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 Access

An instruction that requests a byte forms a byte pointer from the contents of an accumulator. A byte pointer consists of the effective address of the word containing the byte and a byte indicator (BI) as diagrammed below.

Byte Pointer Format



If the byte indicator is 0, the most significant byte (bits 0-7) is accessed. If it is 1, the least significant byte (bits 8-15) is accessed.

Using the byte pointer, the processor accesses the word and then the byte, as shown in Figure 2.1.

> **NOTE:** *Indirect addressing cannot be used for byte access.*

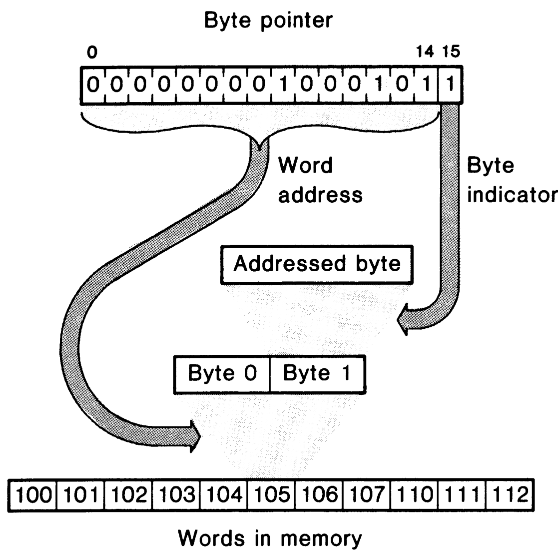
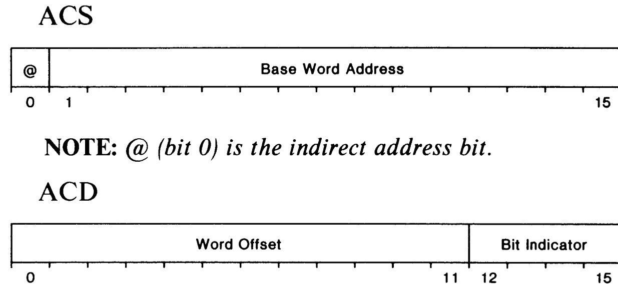


Figure 2.1 Byte addressing

Bit Access

An instruction that accesses a bit, such as a *Set Bit to One* (BTO), forms a word pointer and a bit indicator from the contents of two accumulators (ACS and ACD). The word pointer consists of a base word address in ACS and a word offset in ACD as diagrammed below. The bit indicator is located in the least significant bits of ACD as shown in the diagram.

Bit Pointer Format



NOTE: *@ (bit 0) is the indirect address bit.*

The processor calculates the value of the word pointer by adding the word offset (an unsigned integer) to the base word address. This value is the effective address of the desired word. If the indirect bit (bit 0 of ACS) is one, the base word address is an indirect address and the processor resolves the indirect chain before adding the word offset.

Using this effective address, the processor accesses the word and locates the bit using the bit indicator, as shown in Figure 2.2. The bit indicator specifies the bit position (0-15).

If the instruction specifies ACS and ACD as the same accumulator, the processor assumes the base word address is zero and uses the word offset as the absolute (direct) address of the word.

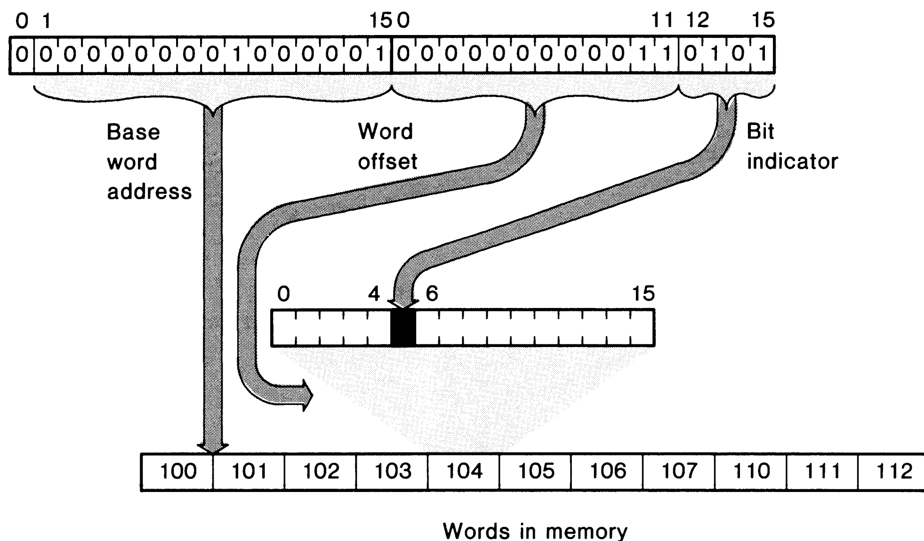


Figure 2.2 Bit addressing

Memory Protection

While executing a memory reference instruction in computers with address translation, the processor checks for indirect addressing violations and for the validity of the memory reference.

If the processor detects such an error, an indirect or protection fault occurs before the execution of the next instruction. For details on the protection mechanisms and fault servicing, refer to Chapter 7, "Memory and System Management."

Stack Management

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 one 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 example, the program can store the contents of up to four fixed-point accumulators on a stack with the *Push Accumulator* (PSH) instruction or restore the contents of the accumulators with a *Pop Accumulator* (POP) instruction. Other instructions allow the program to store a return block on the stack for a subroutine call, an I/O interrupt request, or a fault. Still other instructions retrieve the return block from the stack upon return from the call, interrupt, or fault handler.

Stack Operations

The processor manages the stack parameters for the current user process with three reserved memory locations (40_8 through 42_8). The program can modify the contents of the stack parameters with instructions that move data between an accumulator and the appropriate reserved location.

Figure 2.3 shows the stack parameters. Items (1) and (2), respectively, identify the upper stack limit and the stack base, which define locations that the stack occupies. Items (3) and (4), respectively, identify the stack pointer and the frame pointer, which address the data in the stack.

Stack Base

The *stack base* is the address of the first word in the stack and thus defines the lower limit of the stack. When the program sets up a stack, it must set the stack pointer to the word below the stack base. The standard stack base is 400_8 since processors that detect stack underflow use this address to define a stack underflow condition.

Refer to the subsequent section "Stack Protection" for more information on stack underflow.

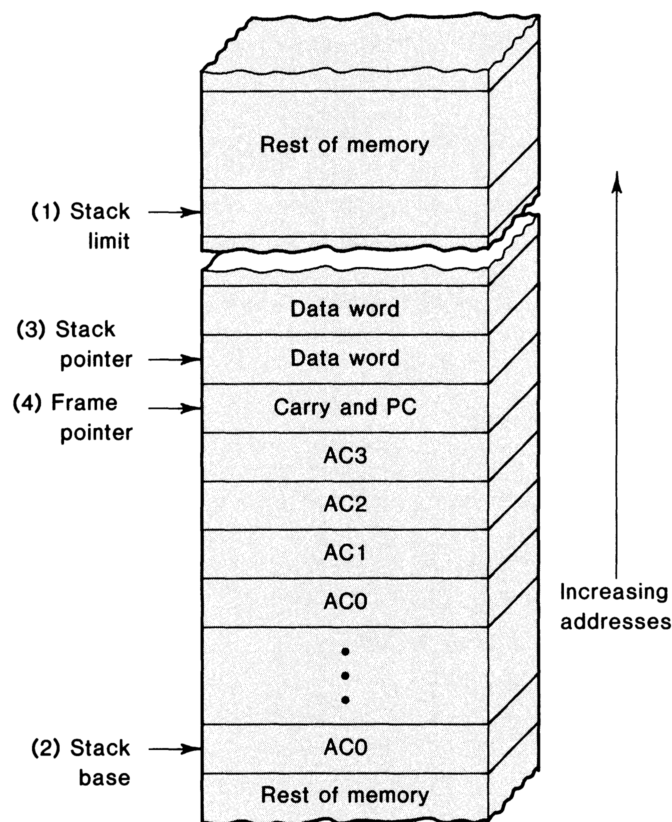


Figure 2.3 Parameters for a typical stack

Stack Limit

The *stack limit* defines the upper limit of the stack. To ensure that stack overflow protection functions correctly, the program must set the stack limit to a given number of words below the address of the last word in the stack. Reserved location 42_8 contains the value of the stack limit.

Adhering to the following rules will ensure proper overflow protection:

- For programs using floating-point instructions, set the stack limit to 23 less than the address of the last location in the stack.
- For programs not using floating-point instructions, set the stack limit to 10 less than the address of the last location in the stack.

The processor uses the stack limit to detect stack overflow conditions. Refer to the subsequent section "Stack Protection," for more information on stack overflow.

Stack Pointer

The *stack pointer* addresses the highest location that the stack thus far has reached. Reserved location 40_8 contains the current value of the stack pointer. When the program sets up a stack, it must set the stack pointer's value to one less than the stack base's value. After the stack pointer's value has been set, each time the processor pushes a word onto the stack, it increments the stack pointer by one; each time the processor pops a word from the stack, it decrements the stack pointer by one. Thus, the stack pointer always points to the last element pushed on the stack.

Frame Pointer

The frame pointer defines a reference point in the stack that is unchanged by push and pop operations. When a program sets up a stack, it should initialize the frame pointer with the same value as the stack pointer to preserve the original value of the stack pointer.

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

Stack Parameter Instructions

The instructions listed in Table 2.3 load (initialize) a memory location reserved for a stack parameter with data from an accumulator or store data into an accumulator from a memory location reserved for a stack parameter.

Mnemonic	Name	Action
LDA	*Load Accumulator*	Copies a word from memory into an accumulator
MSP	*Modify Stack Pointer*	Changes the value of the stack pointer and tests for potential overflow
STA	*Store Accumulator*	Copies the contents of an accumulator into memory

Table 2.3 Stack parameter instructions

Stack Data Instructions

The stack data instructions access a word or block of words. All stack instructions increment or decrement the stack pointer. Instructions that access a word modify the stack pointer by one. Instructions that access a block of words modify the stack pointer by two or more, depending upon the size of the data block or return block.

The instructions in Table 2.4 access a word or block of words and the instructions in Table 2.5 access a return block. Although a return block can take several forms, it usually consists of the five words for fixed-point returns (Table 2.6) or eighteen words for floating-point returns (Table 2.7).

Mnemonic	Name	Action
POP	*Pop Multiple Accumulators*	Pops one to four words off the stack and places them in accumulators. Used with PSH.
POPJ	*Pop PC and Jump*	Pops the top word off the stack and places it in the program counter. Used with PSHJ.
PSH	*Push Multiple Accumulators*	Pushes the contents of one to four accumulators onto the stack. Used with POP.
PSHJ	*Push Jump*	Pushes the address of the next sequential instruction onto the stack and loads an address into the program counter. Used with POPJ.

Table 2.4 Stack word access instructions

Mnemonic	Name	Action
FPOP	*Pop Floating-point State*	Pops a standard floating-point return block off the stack. Used with FPSH.
FPSH	*Push Floating-point State*	Pushes a standard floating-point return block onto the stack. Used with FPOP.
POPB	*Pop Block*	Pops a standard fixed-point return block off the stack. Used with XOP or XOP1.
RSTR	*Restore*	Pops a standard fixed-point return block plus stack parameters off the stack. Used with VCT.
RTN	*Return*	Pops a standard fixed-point return block off the stack to return control from subroutines using SAVE at their entry points.
SAVE	*Save*	Pushes a standard fixed-point return block onto the stack and saves more stack storage space. Used with RTN.
VCT	*Vector on Interrupting Device*	Returns the device code of an interrupting device and uses that code as an index into a vector table. In certain modes, also pushes a standard fixed-point return block onto the stack. Used with RSTR.
XOP	*Extended Operation*	Pushes a standard fixed-point return block onto the stack and transfers control to a routine. Used with POPB.
XOP1	*Alternate Extended Operation*	Pushes a standard fixed-point return block onto the stack and transfers control to a routine. Used with POPB.

Table 2.5 Stack return block instructions

Word Number in Block		
Pushed	**Popped**	**Contents**
1	5	AC0
2	4	AC1
3	3	AC2
4	2	AC3 or frame pointer
5	1	Bit 0 = Carry Bits 1-15 = PC (return address)

Table 2.6 Standard fixed-point return block

Word Number in Block		
Pushed	**Popped**	**Contents**
1-2	17-18	FPSR
3-6	13-16	FPAC0
7-10	9-12	FPAC1
11-14	5-8	FPAC2
15-18	1-4	FPAC3

Table 2.7 Standard floating-point return block

Stack Initialization

The program must initialize the stack parameters before performing the first stack operation. The rules for initialization follow.

1. Initialize the stack pointer to the starting address of the stack minus 1. For underflow protection, start the stack at 401_8 and initialize the stack pointer to 400_8. Otherwise, start the stack at a location greater than 401_8. Note that starting a stack at a location greater than 401_8 does not completely disable underflow protection since it is always possible to pop enough words of the stack to cause stack underflow. To completely disable underflow protection, set bit 0 of both the stack pointer and the stack limit to 1. This allows all or a portion of the stack to reside in lower page zero (locations 0-400_8).
2. Initialize the stack limit to a value greater than the stack pointer. For overflow protection, initialize the stack limit as follows.
   - For programs not using floating-point instructions, set the the stack limit to at least 10_{10} less than the last allocated stack address.
   - For programs using floating-point instructions, set the stack limit to at least 23_{10} less than the last allocated stack address.

   To place all or a portion of the stack in lower page zero, set bit 0 of both the stack limit and the stack pointer to 1.
3. Initialize the stack fault address (contents of location 43_8) to the address that contains the routine to handle stack faults.
4. Initialize the frame pointer to the same value as the stack pointer to preserve the original value of the stack pointer.

Figure 2.4 shows sample Data General assembler code for initializing a stack. The stack resides in locations 256_{10} through 355_{10}. The processor detects a stack overflow 12 words before the actual end of the stack. Figure 2.5 illustrates the result of executing the assembler code in Figure 2.4.

```
         .NREL
BASE:    .BLK   88.         ; Reserve 88 words for the stack
ENDZ:    .BLK   12.         ; Reserve 12 words for the stack end
                            ; zone
         .
         .
         .
         LEF    0,BASE-1    ; Initialize
         STA    0,40        ; stack pointer SP and
         STA    0,41        ; frame pointer FP
         LEF    0,ENDZ      ; Initialize stack limit SL for stack
         STA    0,42        ; overflow when SP = BASE+88
         .
         .
         .
         PSH    2,3         ; Store AC2 and AC3 on stack
         .
         .
         .
```

Sample code for initializing a stack DG-25396

Figure 2.4 Sample code for initializing a stack

Stack Protection

All 16-bit ECLIPSE processors provide stack overflow protection; most provide stack underflow protection.

Stack Overflow

Stack overflow occurs when a program pushes data into the memory area outside that allocated for the stack. If stack overflow occurs, data may be pushed into areas which are reserved for other purposes, possibly overwriting other data or instructions.

The processor checks for overflow during each operation that pushes one or more words onto the stack. In most cases, the processor checks for overflow conditions *after* pushing the data onto the stack. However, for a few types of instructions that use the stack (such as SAVE), certain processors check for a possible stack overflow *before* pushing the data onto the stack.

To check for stack overflow, the processor compares the stack pointer contents to the stack limit contents. If the stack pointer contents are greater than the stack limit contents, an overflow condition exists. Chapter 5, "Program Flow Management," describes the servicing of an overflow fault.

Stack Underflow

Stack underflow occurs when a program pops data from the memory area outside that allocated for the stack. If stack underflow occurs, the program is operating with incorrect and unpredictable information, and data instructions may be destroyed.

The processor checks for stack underflow after each operation that pops one or more words off the stack. To check for underflow, the processor compares the the stack pointer contents to 400_8. If the stack pointer contents are less than 400_8 and bit 0 of the stack limit is 0, an underflow condition exists. Refer to Chapter 5, "Program Flow Management," for information on servicing underflow faults.

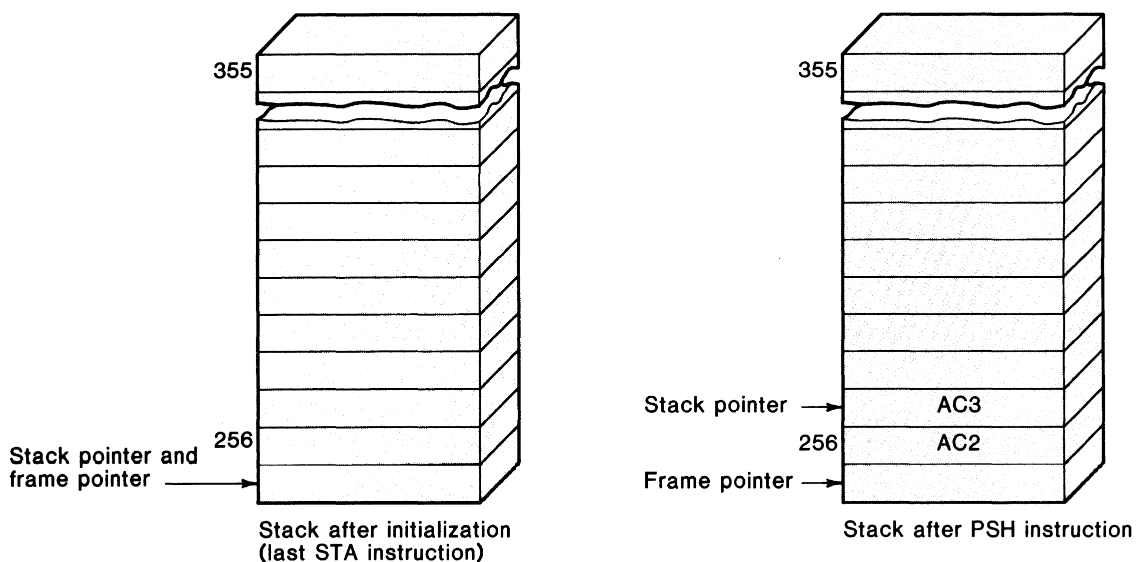


Figure 2.5 Stack operations example

Chapter 3

Fixed-point Computation

All 16-bit real-time ECLIPSE computers can manipulate data using fixed-point operations. With these operations, the processor can add, subtract, multiply and divide fixed-point data. Following the operation, the processor can shift the contents of a fixed-point accumulator and then skip on a condition that results from the computation or shift. Finally, the processor can store the result in a fixed-point accumulator or memory.

Data Formats

Although the processor does not intrinsically distinguish one data type from another, the fixed-point instructions require different types of data as operands. Fixed-point instructions operate on four basic data types — arithmetic, logical, decimal, and byte. Each data type has its own format.

Arithmetic Data

Fixed-point arithmetic instructions operate on arithmetic data expressed as two's complement binary integers. These integers can be signed or unsigned 16-bit (single-precision) or 32-bit (double-precision) numbers.

An unsigned integer uses all the bits to represent the magnitude. A signed integer uses two's complement numbers to distinguish between positive and negative values. In a two's complement number, the left most or most significant bit is used as the sign bit (S). If the sign bit is 0, the number is positive. If the sign bit is 1, the number is negative. Table 3.1 gives the ranges of these numbers.

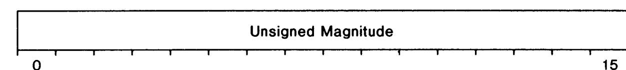
Data Form	Single Precision	Double Precision
Unsigned	0 to 65,535	0 to 4,294,967,295
Signed	−32,768 to +32,767	−2,147,483,648 to +2,147,483,647

Table 3.1 Range of single- and double-precision numbers

The accumulator formats for arithmetic binary data are diagrammed below.

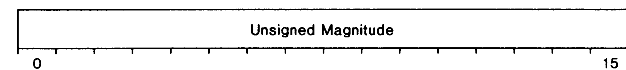
Unsigned Single-precision Format

Word

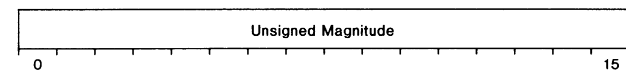


Unsigned Double-precision Format

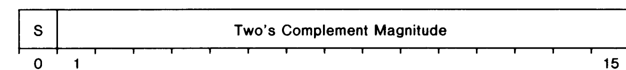
Word 1



Word 2

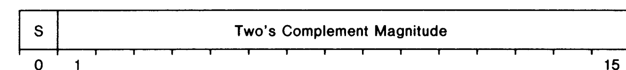


Signed Single-precision Format

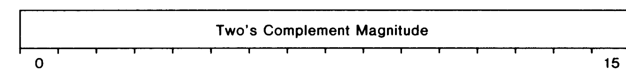


Signed Double-precision Format

Word 1



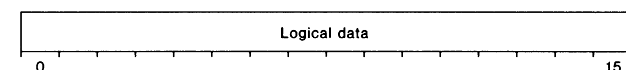
Word 2



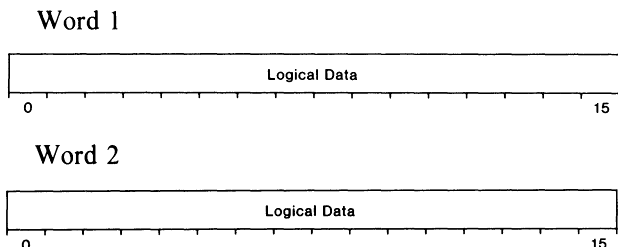
Logical Data

The fixed-point logical instructions operate on data that are 16-bit or 32-bit unstructured binary quantities. These quantities must begin on word boundaries. The accumulator formats for logical data are diagrammed below.

16-bit Logical Format



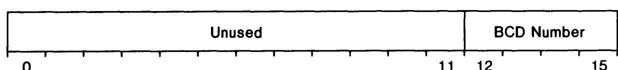
32-bit Logical Format



Decimal Data

The decimal instructions handle unsigned decimal numbers one digit at a time. They require decimal numbers expressed as unsigned packed decimals. For a packed decimal, each digit of the decimal number is expressed as a 4-bit binary coded decimal (BCD). The accumulator format for a BCD decimal digit is diagrammed below.

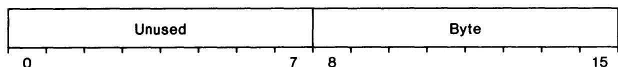
Binary Coded Decimal (BCD) Format



Byte Data

The byte instructions operate on either single 8-bit binary quantities or strings of 8-bit binary quantities (byte strings). Byte strings must start on a word or half-word boundary (bit 8 of the word). The accumulator format for single bytes is diagrammed below.

Byte Format



Common Operations

Many of the fixed-point arithmetic, logical, and decimal instructions provide optional operations that the processor and arithmetic logic unit perform before or after the computation. These common operations can manipulate the carry, shift results, and skip a word on a condition.

Carry Operations

For fixed-point computations, the processor maintains a Carry flag (carry). The Carry flag contains a value of 0 or 1. The carry occurs from the most significant bit (bit 0).

The processor changes the value of the Carry flag as a result of executing a fixed-point computation instruction. Before computing the result, the processor initializes the Carry flag as specified by the instruction's carry option. The carry option leaves the carry unchanged, sets the carry to 0 or 1, or complements the carry.

During an add operation, the processor complements the Carry flag when the addition produces a carry out of the most significant bit. During a subtract operation, the processor sets the Carry flag to 1 when borrowing from the most significant bit. After the computation, the processor modifies the Carry flag, depending on the magnitude of the result, and retains the value of the Carry flag for use with another instruction.

NOTE: *The decimal add and subtract instructions use the Carry flag; however, they do not have the carry option. To initialize the Carry flag before a decimal add or subtract operation, the program must use the carry option of an arithmetic instruction.*

Table 3.5 in the subsequent "Arithmetic Instructions" section lists the instructions that initialize the Carry flag.

Shift Operations

The fixed-point arithmetic instructions and some logical and decimal instructions can shift the result.

For the arithmetic instructions, the shift option can shift the intermediate result one bit position or swap the two bytes. As shown in Figure 3.1, the arithmetic shift can be

- One bit to the left with the Carry flag assuming the value of the most significant bit and the least significant bit assuming the value of the Carry flag;
- One bit to the right with the Carry flag assuming the value of the least significant bit and the most significant bit assuming the value of the the Carry flag;
- A swap of the most significant and the least significant bytes with the Carry flag remaining unchanged.

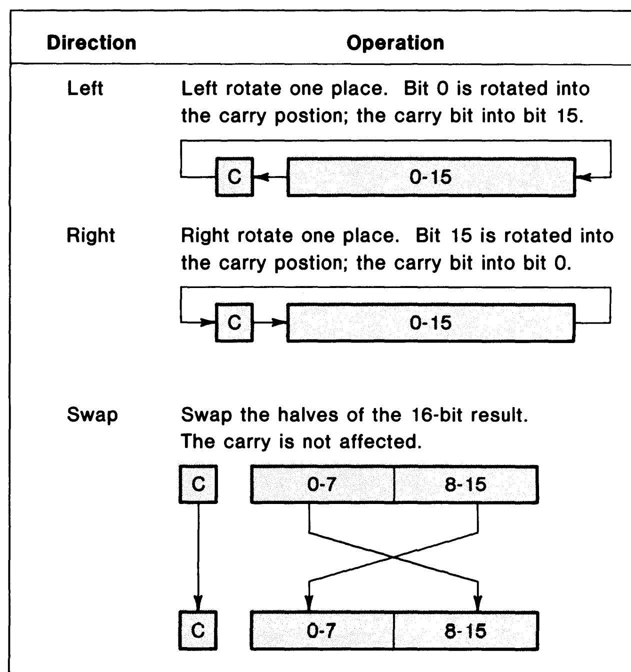


Figure 3.1 Arithmetic shift operation

Table 3.6 in the subsequent "Arithmetic Instructions" section lists the arithmetic shift instructions.

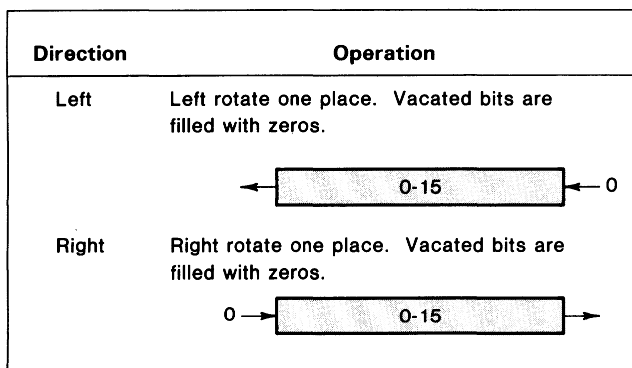
Except for the *AND* and *Complement* (COM) instructions, the logical and decimal instructions do not have a shift option. Instead, the program can shift logical and decimal data using several logical and decimal instructions whose sole function is to shift data. The shift operations performed by these instructions leave the Carry flag unchanged.

The logical shift instructions can shift the data one or more bit positions. As shown in Figure 3.2, the logical shift can be

- One to sixteen bits to the right or left for 16-bit data;
- One to thirty-two bits to the right or left for 32-bit data.

All vacated bits assume a 0 value and all bits shifted out are lost. Table 3.9 in the subsequent "Logical Instructions" section lists the logical shift instructions.

The decimal shift instructions can shift the data one or more hex digit positions (4 bits). As shown in Figure 3.2, the decimal shift can be one to four hex digits to the right or left. As with the logical shift, all vacated bits assume a 0 value and all bits shifted out are lost. Table 3.10 in the subsequent "Decimal Instructions" section lists the decimal shift instructions.



ID-00417

Figure 3.2 Logical shift operation

Skip Operations

With a skip instruction or the skip option of an arithmetic or logical instruction, the processor tests the result of an operation for a specific condition and skips the word or executes the word after the skip instruction.

For an instruction with a skip option, the processor tests the result of the computation during its temporary storage. The processor can then save the result of the computation or ignore it. For fixed-point computation instructions without skip options, the processor stores the result in an accumulator. The program can then test the result with an explicit test and skip on condition instruction.

Whenever a skip occurs, the processor increments the program counter by one and executes the second word after the skip instruction. The program must not use a skip option or instruction to transfer control to the middle of a 32-bit instruction.

Table 3.7 in the subsequent "Arithmetic Instructions" section lists the fixed-point skip instructions.

Move Instructions

Table 3.2 lists the fixed-point data movement instructions. With these instructions the program can

- Move 16-bit or 32-bit fixed-point data between the fixed-point accumulators and memory;
- Move or exchange data between fixed-point accumulators;
- Move blocks of data between memory locations.

Mnemonic	Name	Action
BAM	*Block Move and Add*	Copies words from one location in memory to another, adding a constant to each.
BLM	*Block Move*	Copies words from one location in memory to another.
ELDA	*Extended Load Accumulator*	Copies a memory word to an accumulator.
ESTA	*Extended Store Accumulator*	Copies an accumulator's contents to memory.
LDA	*Load Accumulator*	Copies memory word to an accumulator.
MOV	*Move*	Copies one accumulator's contents to another accumulator.
PSH	*Push Multiple Accumulators*	Stores the contents of one to four accumulators on the stack.
POP	*Pop Multiple Accumulators*	Removes one to four words from the stack and places them in accumulators.
STA	*Store Accumulator*	Copies an accumulator's contents to memory.
XCH	*Exchange Accumulators*	Swaps the contents of two accumulators.

Table 3.2 Fixed-point data movement instructions

Arithmetic Instructions

Tables 3.3 through 3.6 list the fixed-point arithmetic instructions. These instructions allow the program to

- Add, subtract, multiply, and divide arithmetic data (Tables 3.3 and 3.4),
- Initialize the Carry flag (Table 3.5),
- Shift a word right, left, or swap bytes (Table 3.6),
- Skip a single-word instruction on a condition (Table 3.6).

Mnemonic	Name	Action
ADC	*Add Complement*	Adds the logical complement of an unsigned 16-bit integer in an accumulator to the unsigned 16-bit integer in another accumulator.
ADD	*Add*	Adds an unsigned 16-bit integer in an accumulator to an unsigned 16-bit integer in another accumulator.
ADDI	*Extended Add Immediate*	Adds a signed 16-bit integer to an accumulator's contents.
ADI	*Add Immediate*	Adds an integer from 1 to 4 to an accumulator's contents.
BAM	*Block Move and Add*	Copies words from one memory location to another, adding a constant to each.
INC	*Increment*	Increments an accumulator's contents by 1.
SBI	*Subtract Immediate*	Subtracts an integer from 1 to 4 from an accumulator's contents.
SUB	*Subtract*	Subtracts an unsigned 16-bit integer in an accumulator from an unsigned 16-bit integer in another accumulator.

Table 3.3 Fixed-point addition and subtraction instructions

Mnemonic	Name	Action
DIV	*Unsigned Divide*	Divides an unsigned 32-bit integer in two accumulators by an unsigned 16-bit integer in a third accumulator.
DIVS	*Signed Divide*	Divides a signed 32-bit integer in two accumulators by a signed 16-bit integer in a third accumulator.
DIVX	*Sign Extend and Divide*	Extends the sign of one accumulator into a second accumulator and then performs a DIVS (*Signed Divide*) on the result.
HLV	*Halve*	Divides a signed 16-bit integer in an accumulator by 2.
MUL	*Unsigned Multiply*	Multiplies an unsigned 16-bit integer in an accumulator by an unsigned 16-bit integer in another accumulator and adds the result to an unsigned 16-bit integer in a third accumulator.
MULS	*Signed Multiply*	Multiplies a signed 16-bit integer in one accumulator by a signed 16-bit integer in another accumulator and adds the result to a signed 16-bit integer in a third accumulator.

Table 3.4 Fixed-point multiplication and division instructions

Mnemonic	Name	Action
ADC	*Add Complement*	Optionally initializes carry and adds the complement of an unsigned 16-bit integer in an accumulator to an unsigned 16-bit integer in another accumulator.
ADD	*Add*	Optionally initializes carry and adds an unsigned 16-bit integer in an accumulator to an unsigned 16-bit integer in another accumulator.
AND	*AND*	Optionally initializes carry and forms the logical AND of the contents of two accumulators.
COM	*Complement*	Optionally initializes carry and forms the logical complement of an accumulator's contents.
INC	*Increment*	Optionally initializes carry and increments an accumulator's contents by 1.
MOV	*Move*	Optionally initializes carry and copies an accumulator's contents to another accumulator.
NEG	*Negate*	Optionally initializes carry and forms the two's complement of an accumulator's contents.
SUB	*Subtract*	Optionally initializes carry and subtracts an unsigned 16-bit integer in an accumulator from an unsigned 16-bit integer in another accumulator.

Table 3.5 Fixed-point initialize carry instructions

Mnemonic	Name	Action
ADC	*Add Complement*	Adds the logical complement of an unsigned 16-bit integer in an accumulator to an unsigned 16-bit integer in another accumulator and optionally shifts and/or skips.
ADD	*Add*	Adds an unsigned 16-bit integer in an accumulator to a 16-bit integer in another 16-bit accumulator and optionally shifts and/or skips.
DSZ	*Decrement and Skip if Zero*	Decrements a memory word's contents by 1 and skips if the value is 0.
EDSZ	*Extended Decrement and Skip if Zero*	Decrements a memory word's contents by 1 and skips if the value is 0.
EISZ	*Extended Increment and Skip if Zero*	Increments a memory word's contents by 1 and skips if the value is 0.
INC	*Increment*	Increments an accumulator's contents by 1 and optionally shifts and/or skips.
ISZ	*Increment and Skip if Zero*	Increments a memory word's contents by 1 and skips if the value is 0.
MOV	*Move*	Copies an accumulator's contents to another accumulator and optionally shifts and/or skips.
NEG	*Negate*	Forms the two's complement of an accumulator's contents and optionally skips.
SGT	*Skip if ACS is Greater than ACD*	Skips if the signed two's complement in the source accumulator is greater than the signed two's complement in the destination accumulator.
SGE	*Skip is ACS is Greater than or Equal to ACD*	Skips if the signed two's complement in the source accumulator is greater than or equal to the signed two's complement in the destination accumulator.
SUB	*Subtract*	Subtracts two unsigned 16-bit integers, both in accumulators, and optionally shifts and/or skips.

Table 3.6 Fixed-point shift and skip instructions

Logical Instructions

Tables 3.7 and 3.8 list the logical instructions. With these instructions the program can

- Perform logical computations (Table 3.7),
- Logically shift data (Table 3.8),
- Skip a single-word instruction on a condition (Table 3.8).

Mnemonic	Name	Action
ANC	AND *with Complemented Source*	ANDs the logical complement of an unsigned 16-bit integer in an accumulator to an unsigned 16-bit integer in another accumulator.
AND	AND	Logically ANDs the contents of two accumulators.
ANDI	AND *Immediate*	Logically ANDs an accumulator's contents with a 16-bit number.
COB	*Count Bits*	Counts the number of ones in an accumulator and adds it to the contents of another accumulator.
COM	*Complement*	Logically complements an accumulator's contents.
DLSH	*Double Logical Shift*	Logically shifts the 32-bit contents of two accumulators left or right.
IOR	*Inclusive* OR	Inclusively ORs the contents of two accumulators.
IORI	*Inclusive* OR *Immediate*	Inclusively ORs an accumulator's contents with a 16-bit number.
LOB	*Locate Lead Bit*	Counts the number of most significant zeros in an accumulator and adds it to the contents of another accumulator.
LRB	*Locate and Reset Lead Bit*	Sets the leading one bit (most significant one bit) in an accumulator to 0 and performs an LOB operation.
LSH	*Logical Shift*	Logically shifts the contents of an accumulator left or right.
NEG	*Negate*	Forms the two's complement of an accumulator's contents.
SNB	*Skip on Nonzero Bit*	Skips if the addressed bit is 1.
SZB	*Skip on Zero Bit*	Skips if the addressed bit is 0.
SZBO	*Skip on Zero bit and Set to One*	If the addressed bit is 0, sets it to 1 and skips.
XOR	*Exclusive* OR	Exclusively ORs the contents of two accumulators.
XORI	*Exclusive* OR *Immediate*	Exclusively ORs an accumulator's contents with a 16-bit number.

Table 3.7 Logical instructions

Mnemonic	Name	Action
AND	AND	Logically ANDs the contents of two accumulators and optionally shifts and/or skips.
COM	*Complement*	Logically complements an accumulator's contents and optionally shifts and/or skips.
DLSH	*Double Logical Shift*	Logically shifts the 32-bit contents of two accumulators left or right.
LSH	*Logical Shift*	Logically shifts the contents of an accumulator left or right.
NEG	*Negate*	Forms the two's complement of an accumulator's contents and optionally shifts and/or skips.
SNB	*Skip on Nonzero Bit*	Skips if the addressed bit is 1.
SZB	*Skip on Zero Bit*	Skips if the addressed bit is 0.
SZBO	*Skip on Zero Bit and Set to One*	If the addressed bit is 0, sets it to 1 and skips.

Table 3.8 Logical shift and skip instructions

Decimal/Byte Instructions

Tables 3.9 and 3.10 list the decimal and byte instructions. These instructions allow the program to

- Add and subtract decimal numbers (Table 3.9):
- Shift a single or double word to the right or left (Table 3.9).
- Transfer bytes between fixed-point accumulators and memory (Table 3.10);
- Move strings of bytes between memory locations with various control options (Table 3.10).

Mnemonic	Name	Action
DAD	*Decimal Add*	Adds a 4-bit binary coded decimal in one accumulator to a 4-bit binary coded decimal in another accumulator.
DSB	*Decimal Subtract*	Subtracts a 4-bit binary coded decimal in one accumulator from a 4-bit binary coded decimal in another accumulator.
DHXL	*Double Hex Shift Left*	Logically shifts the 32-bit contents of two accumulators left 1 to 8 hex digits.
DHXR	*Double Hex Shift Right*	Logically shifts the 32-bit contents of two accumulators right 1 to 8 hex digits.
HXL	*Hex Shift Left*	Logically shifts an accumulator's contents left 1 to 4 hex digits.
HXR	*Hex Shift Right*	Logically shifts an accumulator's contents right 1 to 4 hex digits.

Table 3.9 Decimal and hex shift instructions

Mnemonic	Name	Action
ELDB	*Extended Load Byte*	Copies a byte from memory to an accumulator.
ESTB	*Extended Store Byte*	Copies a byte from an accumulator to memory.
LDB	*Load Byte*	Same as ELDB, except with a shorter displacement for specifying the memory address.
STB	*Store Byte*	Same as ESTB, except with a shorter displacement for specifying the memory address.
CMP	*Character Compare*	Compares two strings of bytes in memory.
CMT	*Character Move Until True*	Copies a string of bytes from one memory area to another until a delimiter is encountered.
CMV	*Character Move*	Copies a string of bytes from one memory area to another.
CTR	*Character Translate and Move or Compare*	Translates a string of bytes from one data representation to another and either copies the string to another memory area or compares it with a second translated string.

Table 3.10 Byte movement instructions

Chapter 4

Floating-point Computation

Most 16-bit real-time ECLIPSE processors can manipulate data using floating-point operations. With these operations the processor can add, subtract, multiply and divide floating-point data. Following the computation, the processor can convert a floating-point value to a different data format and skip on a condition that results from the computation or conversion. Finally, the processor can store the result in a floating-point accumulator or memory.

Floating-point Data

Floating-point instructions require normalized, signed magnitude numbers. The numbers contain either 32 bits (single precision) or 64 bits (double precision). They must begin on word boundaries and be within the value range of $5.4\text{x}10^{-78}$ to $7.2\text{x}10^{75}$.

A floating-point number consists of the following three fields:

- *Sign bit* which is 0 for a positive number and 1 for a negative number.
- 7-bit *exponent* which is an unsigned integer equal to 64_{10} greater than the true value of the exponent (excess-64 representation). Thus, if the exponent field is zero, the true value of the exponent is -64_{10}; if the exponent field is 64_{10}, the true value of the exponent is 0; and if the exponent field is 128_{10}, the true value of the exponent is 64_{10}.
- 24-bit (single precision) or 56-bit (double precision) *mantissa* which is an unsigned fraction. For a normalized number, the range of the mantissa is

  $1/16$ to $1\text{-}2^{-24}$ for single precision

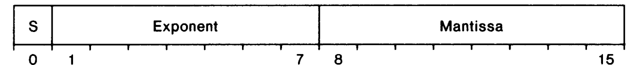
  $1/16$ to $1\text{-}2^{-56}$ for double precision

The floating-point *Normalize* (FNOM) instruction normalizes raw floating-point data, which may or may not be normalized. In addition, if a mantissa equals 0, the processor expects it to be a true zero. A true zero exists when the sign bit, the exponent, and the mantissa all equal 0. (All 32 or 64 bits equal 0.)

The accumulator formats for floating-point numbers follow.

Single-precision Format

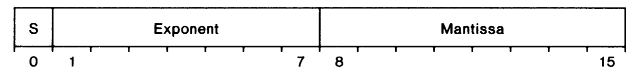
Word 1



Word 2

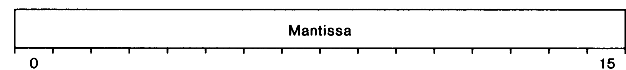
Double-precision Format

Word 1

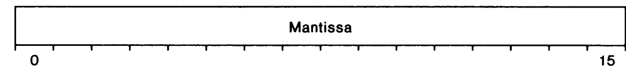


Word 2

Word 3



Word 4



Floating-point Operations

The processor performs a floating-point computation by executing a floating-point instruction. In executing the instruction, the processor

- Appends a guard digit to each operand;
- Aligns the mantissas for addition and subtraction;
- Calculates the intermediate result and normalizes it;
- Truncates the intermediate result;
- Stores the final result in a floating-point accumulator or memory.

Guard Digits

To increase the accuracy of the result, the processor appends a *guard digit* to the operands of both mantissas before performing any arithmetic calculations. A guard digit is one hex digit (four bits) that initially has a zero value.

Mantissa Alignment

Before performing a floating-point addition or subtraction operation, the processor aligns the mantissas of the operands. The processor aligns the smaller mantissa to the larger mantissa by taking the absolute value of the difference between the two exponents. If the difference is nonzero, the processor shifts the mantissa with the smaller exponent right one hex digit at a time until the difference equals zero or until all the significant digits of the mantissa have been shifted out. The mantissas are aligned when the the difference between exponents is zero.

If the processor shifts out the significant digits, the operation is equivalent to adding a zero to the number with the larger exponent. To shift out the significant digits, the processor must shift at least 7 hex digits for single precision and at least 15 digits for double precision.

Calculation and Normalization

The processor performs the floating-point arithmetic operation, determines sign of the intermediate result using the rules of algebra, and then normalizes the result. The processor normalizes the intermediate mantissa by shifting it left one hex digit (four bits) at a time until the most significant hex digit has a nonzero value. 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 as hex digits are shifted left.

Truncation and Storage

After normalizing the intermediate result, the processor truncates it by removing the guard digit. The processor stores the truncated (final) result in the specified memory location or floating-point accumulator and then checks for possible exponent underflow or overflow. If no underflow or overflow exists, instruction execution ends. If an underflow or overflow exists, the value of the exponent is either 128_{10} too large (underflow) or 128_{10} too small (overflow), and the processor sets the appropriate error flag in the floating-point status register.

Data Conversion Instructions

Table 4.1 lists the floating-point conversion instructions. With these instructions, the program can convert and move data between fixed-point and floating-point accumulators, convert a mixed number to a fraction, and scale a floating-point number.

Mnemonic	Name	Action
FAB	*Absolute Value*	Forms the absolute value of the contents of a floating-point accumulator.
FEXP	*Load Exponent*	Loads an exponent from a fixed-point accumulator into a floating-point accumulator.
FFAS	*Fix to AC*	Converts the integer portion of the contents of a floating-point accumulator to a signed integer and copies it to a fixed-point accumulator.
FFMD	*Fix to Memory*	Converts the integer portion of the contents of a floating-point accumulator to a signed integer and copies it to memory.
FINT	*Integerize*	Converts the contents of a floating-point accumulator to an integer and normalizes the result.
FLAS	*Float from AC*	Converts a signed integer from a fixed-point accumulator to floating-point format and copies it to a floating-point accumulator.
FLMD	*Float from Memory*	Converts a 32-bit signed integer in memory to floating-point format and copies it to a floating-point accumulator.
FNEG	*Negate*	Changes the sign of the contents of a floating-point accumulator.
FNOM	*Normalize*	Normalizes the contents of a floating-point accumulator.
FRH	*Read High Word*	Copies the most significant word from a floating-point accumulator to a fixed-point accumulator.
FSCAL	*Scale Floating-point*	Shifts the mantissa of the contents of a floating-point accumulator and and replaces its exponent.

Table 4.1 Floating-point to binary conversion operations

Move Instructions

Table 4.2 lists the floating-point data movement instructions. With these instructions the processor can move

- Single- or double-precision floating-point data between floating-point accumulators and memory.
- Data between accumulators

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.

Mnemonic	Name	Action
FLDD	*Load Floating-point Double*	Copies four memory words to a floating-point accumulator.
FLDS	*Load Floating-point Single*	Copies two memory words to a floating-point accumulator.
FMOV	*Move Floating-point*	Copies the contents of one floating-point accumulator to another.
FSTD	*Store Floating-point Double*	Copies the contents of a floating-point accumulator to memory.
FSTS	*Store Floating-point Single*	Copies the most significant two words of a floating-point accumulator to memory.

Table 4.2 Floating-point data movement instructions

Addition and Subtraction Instructions

Table 4.3 lists the floating-point addition and subtraction instructions. With these instructions, the program can perform single and double-precision addition and subtraction as described below.

Addition

For floating-point addition, the processor first aligns the mantissas of the two operands. After mantissa alignment, 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 using 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 has a nonzero value, the processor normalizes and truncates the intermediate mantissa to produce the final result. The processor stores this result in the destination floating-point accumulator and leaves the contents of the source accumulator unchanged.

If incrementing the exponent produces an exponent overflow, the processor sets the Exponent Overflow flag (OVF) in the floating-point status register to one and terminates instruction execution. If there is no mantissa overflow, but the intermediate mantissa contains all zeros, the processor stores a true zero in memory or in a floating-point accumulator.

Subtraction

For floating-point subtraction, the processor temporarily complements the sign of the source mantissa and then performs a floating-point addition. Upon completion, the processor stores the final result in the destination floating-point accumulator and leaves the contents of the source location unchanged.

Mnemonic	Name	Action
FAD	*Add Double* (FPAC to FPAC)	Adds two 64-bit floating-point numbers that reside in floating-point accumulators.
FAS	*Add Single* (FPAC to FPAC)	Adds two 32-bit floating-point numbers that reside in floating-point accumulators.
FAMD	*Add Double* (memory to FPAC)	Adds one 64-bit floating-point number in memory to another 64-bit floating-point number in a floating-point accumulator.
FAMS	*Add Single* (memory to FPAC)	Adds one 32-bit floating-point number in memory to another 32-bit floating-point number in a floating-point accumulator.
FSD	*Subtract Double* (FPAC from FPAC)	Subtracts two 64-bit floating-point numbers that reside in floating-point accumulators.
FSMD	*Subtract Double* (memory from FPAC)	Subtracts a 64-bit floating-point number in memory from a 64-bit floating-point number in a floating-point accumulator.
FSMS	*Subtract Single* (memory from FPAC)	Subtracts a 32-bit floating-point number in memory from a 32-bit floating-point number in a floating-point accumulator.
FSS	*Subtract Single* (FPAC from FPAC)	Subtracts two 32-bit floating-point numbers that reside in floating-point accumulators.

Table 4.3 Floating-point addition and subtraction instructions

Multiplication and Division Instructions

Table 4.4 lists the floating-point multiplication and division instructions. With these instructions the program can perform single- and double-precision multiplication and division as described below.

Multiplication

For floating-point multiplication, the processor multiplies one floating-point mantissa by the other floating-point mantissa, producing an intermediate 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, truncates it, and stores the final result.

Division

For floating-point division, the source location contains the divisor, and the destination location contains the dividend. The processor tests the divisor for zero. If the divisor is zero, the processor sets the Divide by Zero flag (DVZ) to one in the floating-point status register, 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.

When the mantissas align, the processor performs the following actions.

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

Skip Instructions

Table 4.5 lists the floating-point skip instructions. These instructions allow the program to test the result of a floating-point operation for a specific condition and (except for the *Floating-point Compare* instruction, FCMP) to skip or execute the word after the skip instruction.

With the FCMP instruction, the program can compare two floating-point numbers and set the Zero or Nonzero flags in the floating-point status register to reflect the relationship. The program can then use the FSGT, FSEQ, and FSLT skip instructions to test the status flags.

When a skip occurs, the processor increments the program counter by one and executes the second word after the skip instruction. The program must not use a skip instruction to transfer control to the middle of a 32-bit instruction.

Mnemonic	Name	Action
FDD	*Divides Double* (FPAC by FPAC)	Divides a 64-bit floating-point number in a floating-point accumulator by another 64-bit floating point number in a floating-point accumulator.
FDS	*Divide Single* (FPAC by FPAC)	Divides a 32-bit floating-point number in a floating-point accumulator by another 32-bit floating-point number in a floating-point accumulator.
FDMD	*Divide Double* (Memory by FPAC)	Divides a 64-bit floating-point number in memory by another 64-bit floating-point number in a floating-point accumulator.
FDMS	*Divide Single* (Memory by FPAC)	Divides a 32-bit floating-point number in memory by another 32-bit floating-point number in a floating-point accumulator.
FHLV	*Halve*	Divides a 64-bit floating-point number in a floating-point accumulator by 2.
FMD	*Multiply Double* (FPAC by FPAC)	Multiplies two 64-bit floating-point numbers that reside in floating-point accumulators.
FMMD	*Multiply Double* (Memory by FPAC)	Multiplies a 64-bit floating-point number in memory by a 64-bit floating-point number in a floating-point accumulator.
FMMS	*Multiply Single* (Memory by FPAC)	Multiplies a 32-bit floating-point number in memory by a 32-bit floating-point number in a floating-point accumulator.
FMS	*Multiply Single* (FPAC by FPAC)	Multiplies two 32-bit floating-point numbers that reside in floating-point accumulators.

Table 4.4 Floating-point multiplication and division instructions

Mnemonic	Name	Action[1]
FCMP	*Compare Floating-point*	Compares the contents of two floating-point accumulators and updates the Negative (N) and Zero (Z) flags of the floating-point status register.
FNS	*No Skip*	Executes the next sequential word.
FSA	*Skip Always*	Skips the next sequential word.
FSEQ	*Skip on Zero*	Skips if the Zero (Z) flag of the floating-point status register is 1.
FSGE	*Skip on Greater than or Equal to Zero*	Skips if the Negative (N) flag of the floating-point status register is 0.
FSGT	*Skip on Greater than Zero*	Skips if both the Zero (Z) and Negative (N) flags of the floating-point status register are 0.
FSLE	*Skip on Less than or Equal to Zero*	Skips if either the Zero (Z) or Negative (N) flag of the floating-point status register is 1.
FSLT	*Skip on Less than Zero*	Skips if the Negative (N) flag of the floating-point status register is 1.
FSND	*Skip on No Zero Divide*	Skips if the Divide by Zero (DVZ) flag of the floating-point status register is 0.
FSNE	*Skip on Nonzero*	Skips if the Zero (Z) flag of the floating-point status register is 0.
FSNER	*Skip on No Error*	Skips if the Any (ANY) flag of the floating-point status register is 0.
FSNM	*Skip on No Mantissa Overflow*	Skips if the Mantissa Overflow flag (MOF) of the floating-point status register is 0.
FSNO	*Skip on No Overflow*	Skips if the Exponent Overflow (OVF) flag of the floating-point status register is 0.
FSNOD	*Skip on No Overflow and No Zero Divide*	Skips if both the Exponent Overflow (OVF) and the Divide by Zero (DVZ) flags of the floating-point status register are 0.
FSNU	*Skip on No Underflow*	Skips if the Exponent Underflow (UNF) flag of the floating-point status register is 0.
FSNUD	*Skip on No Underflow and No Zero Divide*	Skips if both the Exponent Underflow (UNF) and the Divide by Zero (DVZ) flags of the floating-point status register are 0.
FSNUO	*Skip on No Underflow and No Overflow*	Skips if both the Exponent Underflow (UNF) and the Exponent Overflow (OVF) flags of the floating-point status register are 0.

Table 4.5 Floating-point skip instructions

[1]Refer to the next section, "Status and Faults", for information on the floating-point status register.

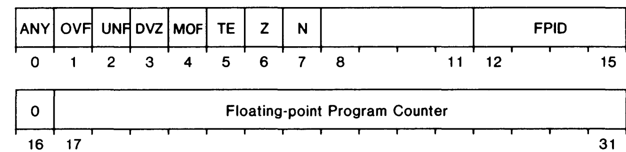
Status and Faults

The processor checks for a floating-point fault and mantissa status after executing a floating-point instruction and stores the result in a 64-bit floating-point status register (FPSR). The format of the floating-point status register is diagrammed below.

When the processor detects a floating-point overflow or underflow fault, it sets the appropriate bits in the status register. At the start of the next floating-point instruction, the processor examines the Trap Enable (TE) flag in the floating-point status register.

If the TE flag is 0, the processor continues normal program execution with the next sequential instruction, and program flow remains unchanged. If the TE flag is 1, the processor disrupts normal program execution to service the routine by performing an indirect jump to the floating-point fault handler. Refer to Chapter 4, "Program Flow Management," for further information on fault handling.

Floating-point Status Register Format



Bits	Name	Contents or Function
0	Any	If 1, one or more of bits 1-4 are set to 1.
1	OVF	Exponent Overflow flag. If 1, exponent overflow occurred. The result is correct except that the exponent is 128 too small.
2	UNF	Exponent Underflow flag. If 1, exponent underflow occurred. The result is correct except that the exponent is 128 too large.
3	DVZ	Divide by Zero flag. If 1, division by zero was attempted. The division operation was aborted and the operands remain unchanged.
4	MOF	Mantissa Overflow flag. If 1, a mantissa overflow occurred.
5	TE	Trap Enable flag. If 1, floating-point traps are enabled and the setting of any of bits 1-4 to 1 causes a floating-point fault.
6	Z	Zero flag. If 1, the result is zero.
7	N	Negative flag. If 1, the result is negative.
8-11	—	Reserved for future use.
12-15	FPID	Floating-point identification code that identifies the floating-point unit.
16	—	Reserved for future use.
17-31	Floating-point	Floating-point program counter. In the event of a floating-point fault, these bits are the logical address of the floating-point instruction that caused the fault.

Status Instructions

Table 4.6 lists the instructions that allow the program access to the floating-point status register. With these instructions the program can initialize and read the register. The program can test the register bits using the skip instructions listed in Table 4.5.

Mnemonic	Name	Action
FCLE	*Clear Errors*	Sets the fault flags (bits 0-4) of the floating-point status register to 0.
FLST	*Load Floating-point Status*	Copies two words from memory to the floating-point status register.
FPOP	*Pop Floating-point State*	Pops a standard floating-point return block off the stack.
FPSH	*Push Floating-point State*	Pushes a standard floating-point return block on the stack.
FSST	*Store Floating-point Status*	Copies the contents of the floating-point status register to memory.
FTD	*Floating-point Trap Disable*	Disables traps on floating-point faults.
FTE	*Floating-point Trap Enable*	Enables traps on floating-point faults.

Table 4.6 Floating-point status register instructions

Chapter 5

Program Flow Management

Programs consist of instruction sequences that reside in main memory. The order in which the processor executes these instructions depends on the 15-bit logical address stored in the program counter (PC). During program execution, the processor fetches and executes the instruction at this logical address.

To address the next instruction for sequential program flow, the processor increments the program counter by

- One when executing a one-word instruction, such as ADD;
- Two when executing a two-word instruction, such as ADDI.

Any of the following events alter sequential program flow:

Executing an *Execute* instruction (XCT),
Executing a jump instruction,
Executing a skip instruction,
Executing a subroutine call or return instruction,
Trapping on a fault,
Detecting an I/O interrupt request.

The following sections describe the XCT, jump, skip, extended operation, and subroutine call or return instructions; and fault handling. Refer to Chapter 5, "Device Management," for I/O interrupt processing.

Execute Instruction (XCT)

The *Execute* instruction (XCT) executes the contents of a fixed-point accumulator as an instruction. If the accumulator does not contain another XCT instruction, after executing the accumulator contents, program flow continues with one of the following logical addresses:

- First address after the XCT instruction if the accumulator contains a single word, nonjump, or nonskip instruction;
- Second address after the XCT instruction if the accumulator contains the first word of a two-word nonjump instruction;
- The effective address, if the accumulator contains a jump or skip instruction.

If the accumulator contains another XCT instruction, the processor waits for an interrupt, provided the accumulator specified by the instruction differs from the accumulator containing the instruction. If the accumulators are the same, then the processor will *loop*, continually executing the second XCT instruction.,

Jump Instructions

A jump instruction loads the effective address into the program counter. Thus, program flow continues at the effective address. A jump instruction does not save a return address. Figure 5.1 illustrates how a jump instruction alters program flow. Table 5.1 lists the jump instructions.

Mnemonic	Name	Action
DSPA	*Dispatch*	Conditionally transfers control to an address selected from a table.
EJMP	*Extended Jump*	Loads an effective address into the program counter.
JMP	*Jump*	Loads an effective address into the program counter.

Table 5.1 Jump Instructions

Skip Instructions

A skip instruction jumps the first word after the skip instruction, and executes the second word as an instruction. To perform the skip, the processors adds two to the program counter. For most skip instructions, the processor first tests a machine condition or status, and based on the test result, it executes the first or second word following the word that contains the skip instruction. Figure 5.1 illustrates how a skip instruction alters program flow.

When using a skip instruction, be sure that the skip does not transfer control to the middle of a two-word instruction. For example, the first two lines of code in Figure 5.2 perform an illegal skip because the program counter contains the address of the FDMD displacement (the second word of the FDMD instruction). The last three lines of code in Figure 5.2 perform the skip properly.

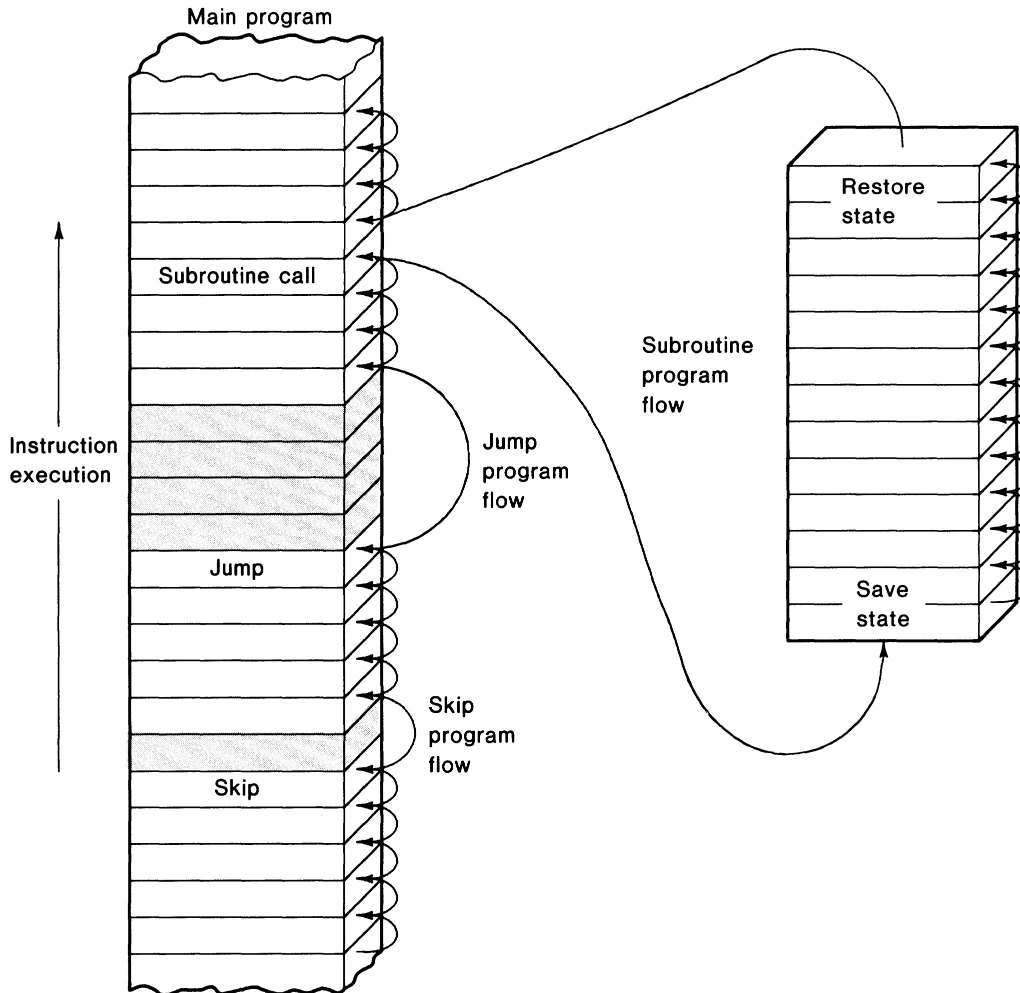


Figure 5.1 Program flow

Skip instructions are available for fixed-point, floating-point, and I/O operations. For more information, refer to

Chapter 3, "Fixed-point Computation," for the fixed-point skip instructions;

Chapter 4, "Floating-point Computation," for the floating-point skip instructions;

Chapter 6, "Device Management," for the I/O skip instructions.

```
        FSEQ                ; Skip on zero result
        FDMD   0,@OPAND     ; Floating-point divide with one
                            ; word displacement instruction
        .
        .
        .
        FSNE                ; Skip on nonzero result and execute
                            ; FDMD
                            ; instruction
        JMP    NEXT         ; Zero result — skip FDMD instruction
        FDMD   0,@OPAND     ; Floating-point divide with one
                            ; word displacement
NEXT:
```

DG-25397

Figure 5.2 Illegal and legal skip instruction sequences

Subroutine Calls and Returns

The *Push Jump* (PSHJ), *Extended Operation* (XOP, XOP1), and *System Call* (SYC) instructions either push a return address or return block onto the stack. Thus, to pass arguments to a subroutine called with one of these instructions, the program should push the arguments onto the stack before calling the subroutine. All subroutine call instructions load the effective address of the subroutine into the program counter. Program flow continues with the effective address in the program counter. Figure 5.1 shows how subroutine calls and returns alter program flow.

A subroutine return instruction pops a return address or return block off the stack, thus restoring the program counter (return address) or the program counter together with the carry and the accumulators (return block). In either case, program flow continues with the instruction following the subroutine call.

Table 5.2 lists the subroutine call, save, and return instructions and Table 5.3 illustrates the relationships between these instructions. Figure 5.3 illustrates the stack operations from the JSR, SAVE, and RTN instructions. Refer to Chapter 2, "Memory Access and Stack Management," for information on stacks and return blocks.

Mnemonic	Name	Action
DSPA	*Dispatch*	Transfers control to an address selected from a dispatch table if the contents of an accumulator are within specified limits.
EJSR	*Extended Jump to Subroutine*	Increments and stores the value of the program counter in an accumulator and then places a new address in the program counter.
JSR	*Jump to Subroutine*	Same as EJSR, except with shorter displacement for specifying new address.
POPB	*Pop Block*	Returns control from a procedure called by a XOP or XOP1 instruction or from an I/O interrupt handler that does not use the stack change facility of the *Vector* (VCT) instruction.
POBJ	*Push Block*	Pops the top word off the stack and places it in the program counter.
PSHJ	*Push Jump*	Pushes the address of the next sequential instruction on the stack and places a new address in the program counter.
RTN	*Return*	Returns control from a subroutine that issues a *Save* instruction (SAVE) at its entry point.
SAVE	*Save*	Saves information required by the Return instruction (RTN).
SYC	*System Call*	Pushes a return block onto the stack and places the address of the system call handler in the program counter.
XOP	*Extended Operation*	Pushes a return block onto the stack and transfers control to one of 32 procedures by indexing an extended operation table.
XOP1	*Alternate Extended Operation*	Same as XOP, but allows transfer to 16 additional procedures. Used in some processors to enter writeable control store, WCS. (Not supported by some 16-bit real-time ECLIPSE processors.)

Table 5.2 Subroutine instructions

Call	Save	Return
DSPA	—	—
EJSR, JSR	SAVE	RTN
PSHJ	—	POPJ
XOP, XOP1	—	POPB

Table 5.3 Sequence and types of subroutine instructions

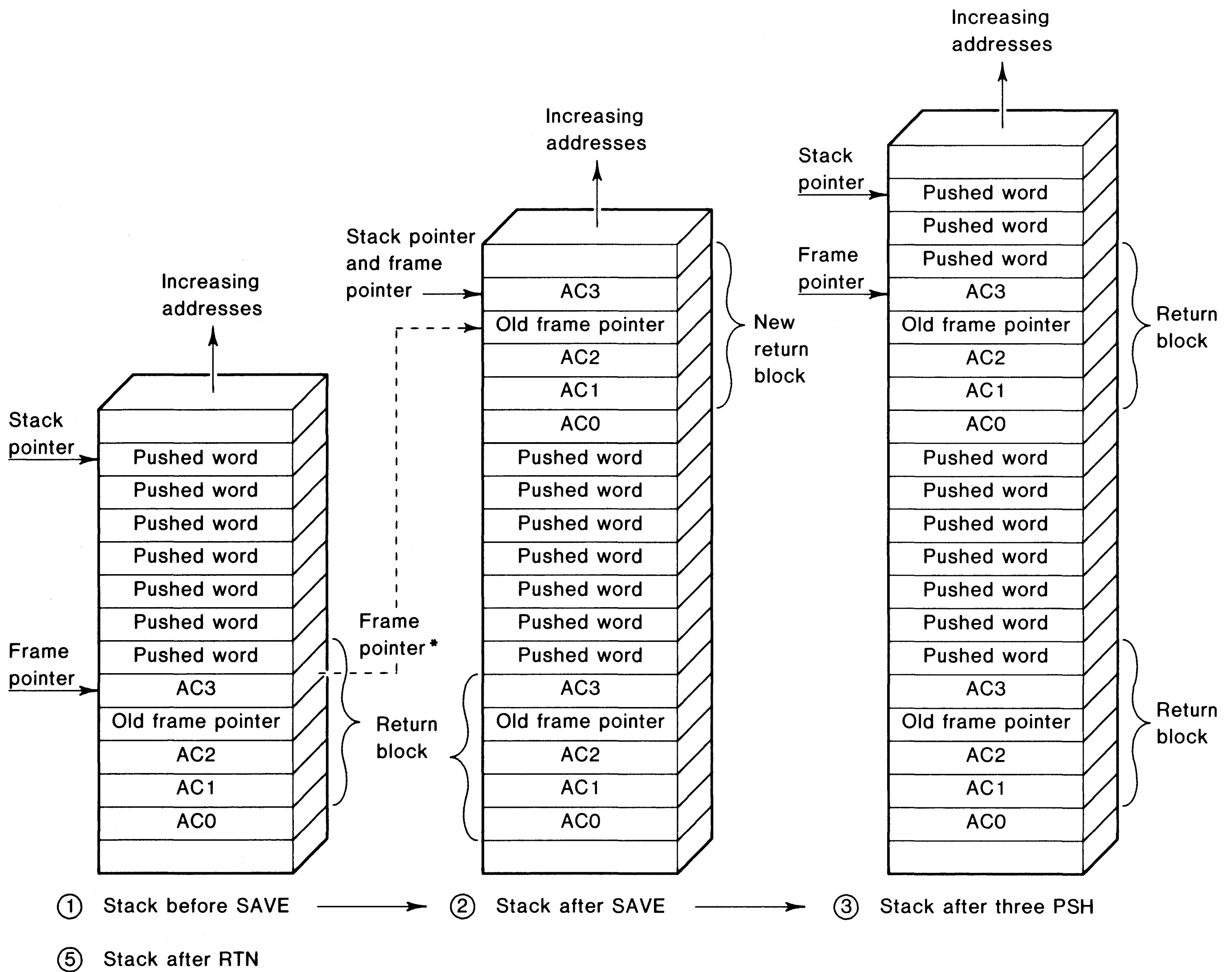
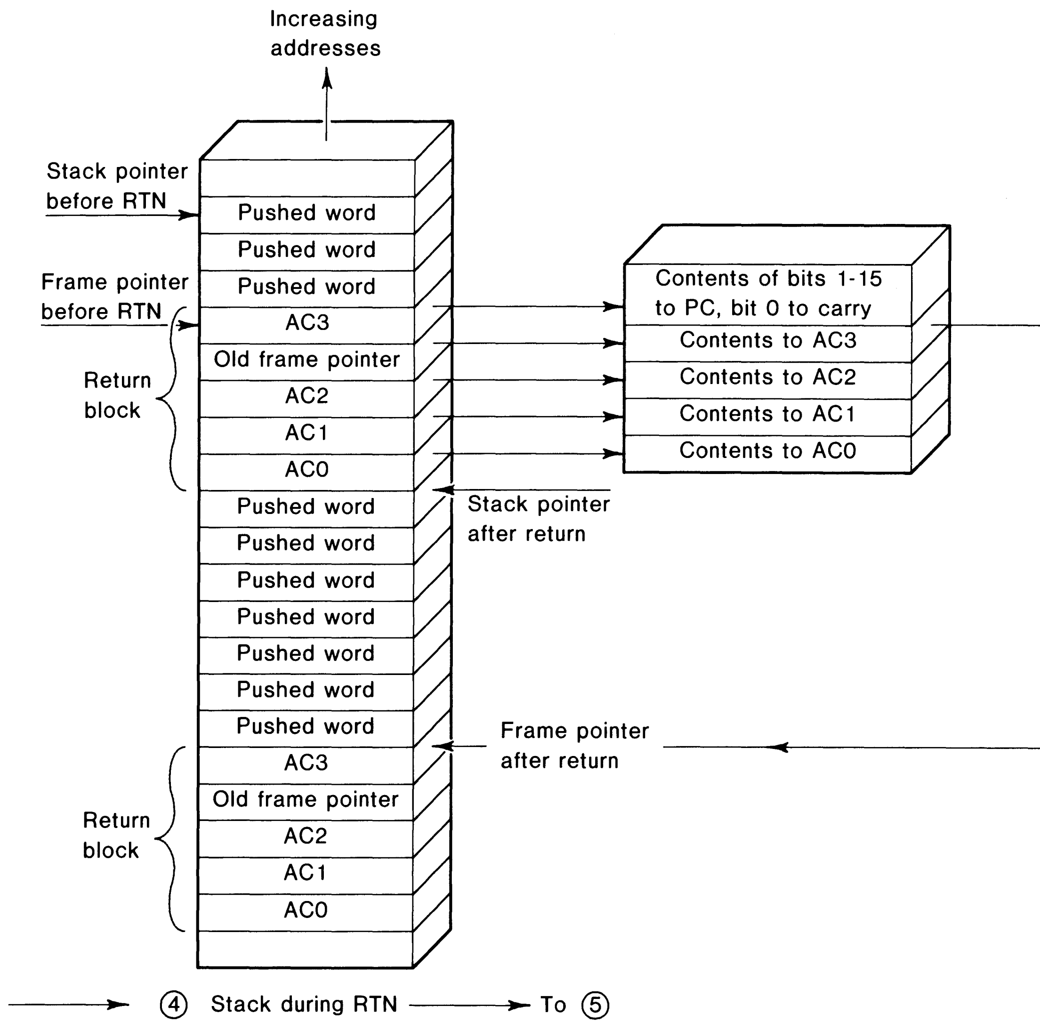


Figure 5.3 Stack operations from the JSR, SAVE, and RTN instructions



④ Stack during RTN → To ⑤

ID-00419

Programming Examples

The four examples that follow show various ways of using the subroutine instructions to link a subroutine program to the main program.

Example 1

The following routine uses the DSPA instruction to process a number of one-letter commands from a user's terminal. The program calls the subroutine with one ASCII character already in bits 8-15 of AC0. CMDTB is the address of a 26-word table in main memory which is preceded by limit words containing the codes for A and Z. ILLCH is the address of a routine to be executed if the user types some character other than a letter. BADCM is the address of a routine to be executed if the user types a letter which has no command function. Command letters may be scattered randomly through the alphabet.

```
        .
        .
        .
                                ; Character now in AC0
        DSPA  0,CMDTB           ; Go to proper routine if user types a
                                ; letter
        JMP   IILCH             ; or here if user did not type ; letter
        .
        .
        .
        101                     ; Lower limit — octal ASCII code for
                                ; "A"
        132                     ; Upper limit — octal ASCII code for
                                ; "B"
                                ; The dispatch table follows
CMDTB:  ACOMD                   ; Address of routine for "A" command
        BCOMD                   ; Address of routine for "B" command
        BADCM                   ; "C" is not a legal command
        BADCM                   ; "D" is not a legal command
        ECMD                    ; Address of routine for "E" command
        .
        .
        .
        ZCOMD                   ; Address of routine for "Z" command
                                ; — the last command
```

With the *Dispatch* instruction, only a single instruction is needed to analyze the character. Without this instruction, testing each letter for validity requires a large number of comparison operations. Although the *Dispatch* instruction requires additional memory for the dispatch table, it saves a considerable amount of time.

Example 2

The following example shows the simplest method of calling a subroutine — using the JSR or EJSR instruction. This instruction increments the contents of the program counter, loads it into AC3, and jumps to the specified address. Thus, AC3 contains the address of the instruction following the JSR or EJSR instruction (the return address). The subroutine returns by executing a jump to the return address in AC3.

The program can pass in-line arguments to the subroutine by placing the values to be passed in the words following the JSR or EJSR instruction. The subroutine can access these words by using AC3 as an index register.

The example calls a subroutine FUNC that reads the number stored in-line as the first argument, performs some functions on it, and places the result in the address specified by the second argument. Note that FUNC must add 2 to AC3 before returning to avoid jumping back into the arguments.

```
        .
        .
        .                       ; Main program code
        JSR   FUNC              ; Call subroutine FUNC
        NUMBR                   ; Number to use
        ANSWR                   ; Address of place for storing result
        .                       ; Rest of main program
        .
        .
ANSWR:                          ; Place for storing result
        .
        .
        .
FUNC:   LDA   0,0,3             ; Load argument into AC0
        .                       ; Process it and place result in AC2
        .
        .
        STA   2,@1,3            ; Store result back in calling program
        ADI   2,3               ; Adjust return address
        JMP   0,3               ; Return to calling program
```

The subroutine linkage with the main program is simple and fast, but it does not support nesting of subroutines or recursive subroutines. While the subroutine does not use any local memory, it does require that AC3 be reserved for the return address.

Example 3

The following example uses the stack facility to improve the efficiency of the subroutine linkage with the main program in several ways. If the program does not need to preserve the accumulators, it can use them to pass arguments, and use the PSHJ instruction to call subroutines and the POPJ instruction to return. Nesting and recursion are possible with the PSHJ/POPJ sequence since the last-in/first-out operation of the stack preserves the return addresses.

This example performs a factorial function for 16-bit unsigned integers using a recursive technique. The first part of the subroutine calls itself recursively, storing data in the stack which the second part uses to calculate the factorial value. The maximum input value that does not cause an overflow is 8.

```
        .
        .
        .                    ; Input argument N in AC0
        PSHJ     FACT        ; Call subroutine FACT which
                             ; outputs result in AC1
        .                    ; (result = 0 if overflow occurs)
        .
        .
FACT:   MOVZR#   0,0,SNR     ; Does input argument = 0 or 1?
        JMP      LTTLE       ; Yes, output is 1
        PSH      0,0         ; No, output=N*(N-1)! so store N in
                             ; stack
        SBI      1,0         ; Decrement argument by 1
        PSHJ     FACT        ; Loop around to put proper
                             ; information on the stack
        POP      2,2         ; Get argument from stack
        SUB      0,0         ; Clear AC0 for integer multiply
        MUL                  ; Multiply N*(N-1)!
        MOV#     0,0,SNR     ; Overflow? (If AC0 is not 0)
        SUB      1,1         ; Yes, signal overflow with 0 result
        POPJ                 ; Return to next address in stack
LTTLE:  SUBZL    1,1         ; Put 1 in AC1
        POPJ                 ; Return to next address in stack
```

Example 4

The following example uses the SAVE and RTN instructions with the JSR instruction to provide a powerful routine linkage with the calling program that supports nesting and recursion with local variables stored on the stack, and automatic saving of accumulators.

This example uses the SAVE instruction to save the accumulators on the stack and reserve five words of the stack for local variables. It then loads the words pointed to by two in-line arguments into the first two words of the local memory. The subroutine returns to the calling program with a RTN instruction, which restores the original condition of the accumulators and the stack.

```
        .
        .
        .
        JSR     SUBRT    ; Call subroutine SUBRT
        .
        .
        .
SUBRT:  SAVE    5        ; Save ACs, reserve 5 words for local
                         ; storage. (AC3 now points to the last
                         ; word pushed onto the stack.)
        LDA     2,0,3    ; Load address of arguments into AC2
        LDA     1,@0,2   ; Get first argument
        STA     1,1,3    ; Store it into first local variable
        LDA     1,@1,2   ; Get second argument
        STA     1,2,3    ; Store it into second local variable
        .
        .
        .
        LDA     1,0,3    ; Get return address
        ADI     2,1      ; Add two because of in-line arguments
        STA     1,0,3    ; Place correct address in frame
                         ; pointer
        RTN              ; Return to calling program
```

Fault Handling

While executing an instruction, the processor performs certain checks on the operation and the data. If the processor detects an error, then a protection, stack, or floating-point fault occurs.

When the processor detects a protection or stack fault, it pushes a return block onto the stack and jumps to the fault handler through the indirect pointer in reserved memory.

If a protection fault occurs while the processor is handling another fault, the processor aborts the current fault and processes the protection fault. Refer to Chapter 7, "Memory and System Management," for protection fault handling.

If an I/O interrupt occurs after detecting a stack or floating-point fault, the processor pushes a fault return block onto the stack, 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 stack or floating-point fault.

To service a stack or floating-point fault, the processor

1. Depending on the fault, adjusts
   - The stack pointer and stack limit to identify the fault when a stack fault occurs. (The subsequent "Stack Faults" section describes these adjustments.)

- Adjusts the error flags in the floating-point status register to identify the fault when a floating-point fault occurs. (The subsequent "Floating-point Faults" section describes these adjustments.)

2. Pushes a fault return block onto the stack. The program counter in the fault return block usually is the address of the instruction causing the fault or points to the instruction immediately following the stack instruction that caused the fault.
3. Checks for stack overflow. If 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 the fault handler should be a POPB instruction for the processor to continue executing the interrupted program.

Stack Faults

A stack operation fault results from a stack underflow or overflow. 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 stack fault.

> **NOTE:** *Most 16-bit real-time ECLIPSE computers detect both stack overflow and underflow; however, some only detect overflow.*

After a stack push operation, the processor compares the contents of the stack pointer to the contents of the stack limit. If the stack pointer value is greater than the stack limit value, the processor detects a stack overflow fault. The program can disable stack overflow detection by setting bit 0 of the stack pointer to 0 and bit 0 of the stack limit to 1.

After a stack pop operation, the processor compares the contents of the the stack pointer to 400_8. If the stack pointer value is less than 400_8 and bit 0 of the stack limit is 0, the processor detects a stack underflow fault. The program can disable stack underflow protection by

- Starting the stack at a location greater than 401_8. If the stack starts at a location greater then 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 stack to cause the stack pointer to wrap around.
- Setting bit 0 of either the stack pointer or stack limit to 1. If bit 0 of the stack pointer or stack limit is set to 1, either all or a part of the stack may reside in lower page zero (the first 256_{10} locations), or the the stack may underflow onto lower page zero without interference from the stack fault handler.

When a stack fault occurs, the processor

1. Sets the stack pointer equal to the stack limit for stack underflow.
2. Sets bit 0 of the stack pointer to 0 and bit 0 of the stack limit to 1. Thus, the stack limit is temporarily larger than the stack pointer, which disables overflow fault detection.
3. Pushes a standard fixed-point return block onto the stack. The program counter in the return block points to the next instruction the processor executes after servicing the fault.
4. Jumps to the stack fault handler through the indirect pointer in reserved memory location 43_8.

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.

The stack fault handler should

1. Determine the exact nature of the stack fault by examining the contents of the stack pointer and the stack limit. Three conditions can occur:
   - If the address in the stack pointer is not greater than the address in the stack limit, the fault is an overflow resulting from a SAVE instruction.
   - If the address in the stack pointer exceeds the address in the stack limit by more than five, the fault is an overflow resulting from a nonSAVE stack instruction.
   - If the address in the stack pointer exceeds the address in the stack limit by exactly five, then the fault is an underflow.
2. Reset bit 0 of the stack pointer and stack limit to their original values.
3. Take any other appropriate action, such as allocating more stack space or terminating the program.
4. Use a POPB instruction as the return instruction.

Table 5.4 lists the instructions that push or pop one or more words onto the stack and gives the number of words required beyond the stack limit for a stack fault return block.

Instruction		Number of Words	
Mnemonic	Name	Pushed or Popped	Beyond Stack Limit
FPOP	*Floating-point Pop*	18	5
FPSH	*Floating-point Push*	18	23
MSP	*Modify Stack Pointer*		
POP	*Pop Multiple Accumulators*	1-4	5
POPB	*Pop Block*	5	5
POPJ	*Pop Program Counter and Jump*	1	5
PSH	*Push Multiple Accumulators*	1-4	6-9
PSHJ	*Push Jump to Subroutine*	1	6
PSHR	*Push Return Address*	1	6
RSTR	*Restore*	9	5
RTN	*Return*	5	5
SAVE	*Save*	5	10

Table 5.4 Words required beyond stack limit for stack fault return block

Floating-point Faults

A floating-point fault results from a floating-point overflow or underflow condition that occurs when the processor attempts division by zero or calculates 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, or MOF) and the ANY flag to 1 in the floating-point status register. Refer to Chapter 4, "Floating-point Computation," for information on these flags.

For the processor to service a floating-point fault, the program must set the floating-point Trap Enable (TE) flag to 1 in the floating-point status register before the processor sets a floating-point fault flag. The program can use the *Floating-point Trap Enable* instruction (FTE) to set the TE flag to 1 and the *Floating-Point Trap Disable* instruction (FTD) to set the TE flag to 0.

If the TE flag is 0 when the processor sets a floating-point fault flag to 1, the processor ignores the overflow or underflow and continues normal program execution with the next sequential instruction.

If the TE flag is 1 when the processor sets a floating-point fault flag to 1, the processor initiates a floating-point overflow or underflow fault at the start of the next floating-point instruction. In this case, the processor

1. Pushes a floating-point return block onto the stack. (The contents of the program counter in the return block are the logical address of the floating-point instruction that caused the fault.)
2. Sets the TE flag to 0.
3. Transfers program control to the floating-point fault handler by jumping through the indirect pointer in reserved memory location 45_8.

The floating-point fault handler should

1. Determine the nature of the fault by examining the floating-point status register.
2. Take any appropriate action.
3. Set the TE flag to 1 to reenable servicing of floating-point faults.
4. Use a *Pop Block* (POPB) instruction as the return instruction.

Chapter 6

Device Management

All 16-bit real-time ECLIPSE processors support devices that transfer data using slow and medium transfer rates. With a programmed I/O facility, the processor transfers a word 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. Most processors can also support devices that transfer data using a high-speed transfer rate. With an optional burst 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, magnetic tape drives, and disks, transfer data with the data channel I/O facility. High-speed disks transfer data with the burst multiplexor channel.

Operating systems usually access a device through a system call that invokes a device handler. This chapter presents basic information for writing an interrupt handler or a device driver, which the program invokes with a system call.

Device Access

The processor accesses a device through any of the I/O facilities with address translation enabled or disabled. For the processor to access a device with address translation enabled, the LEF (Load Effective Address) flag and the I/O Protect flag in the user/DCH address translator status register must be set to 0.

The LEF flag specifies how the processor interprets LEF and I/O instruction opcodes. For instance, in a program module where the processor is executing LEF instructions, this flag must be set to 1 — selecting LEF mode. Thus, the processor interprets and executes the I/O and LEF instructions as LEF instructions. Conversely, in a program module where the processor executes I/O instructions, the LEF flag must be set to 0 — selecting the I/O mode. Thus, the processor interprets I/O and LEF instructions as I/O instructions. Executing I/O instructions requires an additional interpretation of the I/O Protect flag.

The I/O Protect flag enables or disables I/O instruction execution. For instance, in a program module where the processor executes I/O instructions, the I/O Protect flag must be set to 0. If this flag is 1, the processor detects a protection violation when attempting to execute an I/O instruction. Refer to Chapter 7, "Memory and System Management," for protection fault handling.

Before the processor can access a device with the data channel or burst multiplexor channel facility, the program must initialize the device. To do this, the program must specify to the device driver and the device

- The direction of the transfer (read or write);
- The address of the first word to be transferred. The device transmits a word address to a device map table (device address translation table). A device map table is a set of registers that control memory addressing for the data transfer.
- The total number of words to transfer.

The data channel uses the device map table in either mapped mode (address translation enabled) or unmapped mode (address translation disabled). In unmapped mode, the processor passes the word address directly to memory as a physical address.

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

Once the program initializes the device, the transfer takes place in two phases.

1. First, the device driver initializes the device with the
   - Starting word address of the block or subblock to transfer,
   - Number of words to transfer,
   - Direction of transfer.
2. Second, the data channel or burst multiplexor channel facility transfers the data between the device and memory.

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

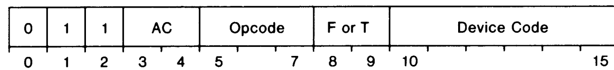
Refer to Chapter 7, "Memory and System Management," for information on the instructions that manipulate device maps. For detailed information on the map registers (map table entries), refer to the assembly language programming manual for the specific processor.

General I/O Instructions

The program controls the devices with I/O instructions. A general set of I/O instructions provide device-independent operations. When issued to a specific device code, these instructions communicate with the specified device to set up data transfers, perform special operations, read status, and initialize and test the device's Busy and Done flags.

The general I/O instructions have the format diagrammed below. Table 6.1 lists the general I/O instructions.

General I/O Instruction Format



Bits	Name	Contents
0-2		011, the binary code indicating an I/O instruction.
3-4	AC	AC field indicating a fixed-point accumulator (0-3). The accumulator contains the data to send or receive from a device.
5-7	Opcode	Opcode field identifying the I/O instruction operation. (Table 6.1 lists the I/O instructions.)
8-9	*f* or *t*	*f* identifies the device flag to change as defined in Table 6.2. *t* identifies the device flag to test as defined in Table 6.3.
10-15	Device Code	Device code field identifying a unique device controller/interface to send or receive data. With this 6-bit code, the of the interrupt system processor can communicate with up to 64_{10} device controllers. Data General's assembler translates a standard three-, four-, or five-letter device mnemonic into a device code. Refer to the assembly language programming manual for a specific processor for a complete list of standard device mnemonics for the corresponding device code.

Mnemonic	Name	Action
DIA*[f]*	*Data in A*	Transfers data from the A buffer of an I/O device to an accumulator
DIB*[f]*	*Data in B*	Transfers data from the B buffer of an I/O device to an accumulator
DIC*[f]*	*Data in C*	Transfers data from the C buffer of an I/O device to an accumulator
DOA*[f]*	*Data out A*	Transfers data to the A buffer of an I/O device from an accumulator
DOB*[f]*	*Data out B*	Transfers data to the B buffer of an I/O device from an accumulator
DOC*[f]*	*Data out C*	Transfers data to the C buffer of an I/O device from an accumulator
NIO*[f]*	*No I/O Transfer*	Changes the state of a device's Busy and Done flags without performing any other operation
SKP*t*	*I/O Skip*	Tests a device's Busy and Done flags and skips on a condition

Table 6.1 General I/O instructions

[1] *The [f] or t defines optional device flag handling as summarized in Tables 6.2 and 6.3, respectively.*

Mnemonic for *f*	Bits 8	9	I/O Device Flag Busy	Done	CPU flag ION[1]
(option omitted)	0	0	No effect	No effect	No effect
S	0	1	Set to 1	Set to 0	Set to 1
C	1	0	Set to 0	Set to 0	Set to 0
P	1	1	Pulses a special control I/O line		No effect

Table 6.2 Device flags for general devices

[1] *ION is the Interrupt On flag of the interrupt system.*

Assembler Mnemonic for *t*	Bits 8	9	I/O Device Flag	CPU Flag [1]
BN	0	0	Test for Busy = 1	Test for ION = 1
BZ	0	1	Test for Busy = 0	Test for ION = 0
DN	1	0	Test for Done = 1	Test for Powerfail = 1
DZ	1	1	Test for Done = 0	Test for Powerfail = 0

Table 6.3 Device flags for skip instruction

[1] *ION is the Interrupt On flag of the interrupt system.*

The device's Busy and Done flags indicate the device state to a device driver. When both flags are 0, the device is idle. To start a device, the device driver issues an I/O instruction to the device with the device flag that sets the Busy flag to 1 and the Done flag to 0. When the device finishes the operation and becomes ready to start another operation, the device sets the Busy flag to 0 and the Done flag to 1.

The Interrupt On (ION) flag controls the device interrupt system. When the ION flag is 0, the processor ignores interrupt requests. When the ION flag is 1, the processor services interrupt requests.

The Powerfail flag indicates the power status to the interrupt system driver. When the Powerfail flag is 0, the processor has detected the proper power voltage ranges. When the Powerfail flag is 1, the processor has detected a powerfail condition.

Interrupt System

The processor and operating system maintain the I/O facilities through a priority 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 an I/O system call, which initializes the device and transfers data by invoking the interrupt system.

NOTE: *The interrupt system, often referred to as the "CPU device," has the mnemonic CPU and the device code* 77_8.

The operating system controls the interrupt system by manipulating an Interrupt On (ION) flag, interrupt mask, and device flags. The Interrupt On flag enables or disables all interrupt recognition. The interrupt mask enables or disables selective device interrupt recognition.

The device flags reside in the device controller. They provide the interrupt communications link between the central processor and the device. By manipulating the flags and the interrupt mask, the operating system can cause the processor to ignore all interrupt requests or to service certain interrupt requests selectively.

Interrupt System Instructions

The interrupt system responds to I/O instructions issued to the processor. The assembler interprets these instructions using either the standard or special I/O instruction format. Device flags cannot be appended to the special format of an interrupt system instruction. Table 6.4 lists both formats of these instructions; the standard format is given in parentheses.

Mnemonic[1]	Name	Action
INTA (DIB *[f] ac*,CPU)	*Interrupt Acknowledge*	Returns the device code of the highest priority device requesting service.
INTDS (NIOC CPU)	*Interrupt Disable*	Disables interrupt system.
INTEN (NIOS CPU)	*Interrupt Enable*	Enables interrupt system.
IORST (DIC*[f] ac*,CPU)	*I/O Reset*	Resets the I/O system.
MSKO (DOB*[f] ac*,CPU)	*Mask Out*	Specifies the interrupt priority mask.
— (SKP*t* CPU)	*CPU Skip*	Tests the ION or Powerfail flag and skips on a condition.
RSTR	*Restore*	Returns control from an I/O service routine called by a Vector On Interrupting Device (VCT) instruction.
VCT	*Vector On Interrupting Device*	Returns the device code of the interrupting device and uses that code as an index into a vector table.

Table 6.4 Interrupt system instructions

[1]*Standard mnemonics are given in parentheses. The [f] or t in the mnemonics defines device handling as summarized in Tables 6.2 and 6.3, respectively.*

Interrupts

If the Interrupt On flag and the interrupt mask enable processor recognition of the interrupt request, the processor services the interrupt. The processor recognizes interrupt requests from devices in order of their priority on the I/O bus. In most cases, I/O bus priority is determined by physical proximity to the processor, with the device controller closest to the processor having the highest priority. To service the 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.

Instruction Interruption

Most instructions are noninterruptible because they require only a minimum of processor execution time. For instructions that require more time, such as the *Block Move* (BLM) instruction, the processor interrupts the executing instruction to service the interrupt. After servicing the interrupt, the processor either restarts or resumes the interrupted instruction. Refer to the assembly language programming manual for a specific processor for further information on interruptible, restartable, and resumable instructions.

Interrupt Mask

A device is associated with one of 16 bits in the interrupt mask. When the bit is 1, the mask blocks an interrupt request to the processor from the device associated with that bit. When the bit is 0, the processor services an interrupt request from the device, provided the interrupt system is enabled. (ION flag is 1.) Since the processor can address more than 16 device controllers, a bit in the interrupt mask can be associated with two or more devices, usually with similar transfer rates. Some devices — the CPU device and the address translators, for instance — are not maskable.

Interrupt Level

The operating system should maintain an interrupt-level count which equals

- Zero for a base-level state — the state when no I/O device interrupts are masked out and no interrupts are being serviced. (User programs run in this state.)
- Nonzero for a non-base-level state — the state when some I/O interrupts are masked out or interrupts are being being processed. (Interrupt handlers operate in this state.)

  Each time the processor responds to an interrupt, the interrupt handler should increment this count by 1 and each time the interrupt handler finishes servicing an interrupt, it should decrement this count by 1.

Interrupt Servicing

In response to an interrupt request, the processor

1. Sets the ION flag to 0 so that no devices can interrupt the first part of the service routine.
2. Disables the address translation (MAP) facilities, if they are enabled. Refer to Chapter 7, "Memory and System Management," for information on the address translation facilities.
3. Places the contents of the updated program counter into physical location 0.
4. Jumps indirectly through location 1 to the the first instruction of the interrupt handler. Location 1 should be the address of the interrupt handler.

The interrupt handler should be re-entrant so that if a device routine is interrupted by a higher priority device, the handler does not lose the information needed to restore the state of the machine. To be re-entrant the interrupt handler must save the contents of location 0 (the return address) and the current priority mask each time it is entered at a higher level.

The interrupt handler should also

1. Check the current interrupt level count. If the count is 0 (base-level state), the interrupt handler should save the user stack parameters and define a stack of its own.
2. Save the state of the processor -- accumulators, carry, program counter (return address).
3. Transfer control to the interrupting device's service routine. Before servicing the device, this routine should
   - Save the current priority mask and establish a new one.
   - Dismiss the interrupt it is servicing by setting the device's Done flag to 0.
   - Enable interrupts.

   After servicing the device, the routine should
   - Disable interrupts.
   - Decrement the interrupt level count by 1 and, if the result is 0 (base-level), restore the user stack.
   - Store the priority mask to the condition it was in when the routine was entered.
   - Restore the state of the processor.
   - Enable interrupts.
   - Return control to the interrupted program (to the return address).

Instead of working with the user stack, the interrupt handler should define its own stack for itself for the following reasons:

- A user stack may not always be defined.
- The user stack pointer may rest just below the stack limit, in which case the interrupt handler would overflow the user stack.

The interrupt handler should change the stack environment whenever a transition is made from base level to non-base level or vice versa. If the interrupt handler is processing an interrupt when another interrupt occurs, the stack environment should not change since it already changed for the first interrupt. The interrupt handler can always push information, which it requires for an easy return to processing the first interrupt, onto the new stack before processing the second interrupt.

Vector Interrupt Processing

The *Vector on Interrupting Device Code* (VCT) instruction simplifies the design of an interrupt handler. It streamlines numerous steps into one instruction with five modes, each suited for a different circumstance.

The simplest mode, similar to the *Interrupt Acknowledge* (INTA) instruction, executes rapidly and does not save information about the processor state at the time of the interrupt. In contrast, the most complex mode executes more slowly than simpler modes and

saves information about the state of the machine upon interruption,

stores the user stack parameters,

creates a new stack,

resets the priority mask.

The mode used in a particular situation depends on the importance of such capabilities as saving the machine state, creating a separate vector stack, and changing the priority mask, as compared to the additional time required for processing the interrupt.

Mode A is for devices that require immediate interrupt service; i.e., unbuffered devices with very short latency times or real-time processes requiring immediate access. This mode executes rapidly and does not save data on the machine state at interruption.

Modes B through E each create a priority structure that permits a device needing immediate service to interrupt the servicing of certain other devices. These modes execute more slowly than mode A.

Modes D and E should be used by the interrupt handler only when operating at base level (not while processing interrupts), since these modes create a new vector stack. The instruction stores the (old) user stack parameters on the new stack. In mode E, it also pushes a return block onto the vector stack to facilitate return to the base level. Once the vector stack has been created, the interrupt handler should not attempt to recreate it if a new interrupt occurs before the one in progress is completely serviced. The interrupt handler can use modes B and C during non-base level operations (while processing other interrupts), since these modes do not create a new stack. Mode C pushes a new return block onto the stack to facilitate return to the previous level.

Refer to the VCT instruction description in Chapter 8, "Instruction Dictionary," for more information.

Chapter 7

Memory and System Management

The processor supports memory management and system management facilities for an operating system. Memory management facilities provide

- Memory allocation and protection by translating logical addresses into physical addresses and controlling access to physical memory,
- Memory integrity by checking and correcting the contents of physical memory.

System management facilities provide

- Information about system status and service faults,
- Special system functions.

Memory Allocation and Protection

The 16-bit real-time ECLIPSE processors support a logical memory size of 64 Kbytes and a physical memory size which varies from 64 Kbytes up to 2048 Kbytes, depending on the memory system. All processors that support more than 64 Kbytes of physical memory also use address translation to store 2 Kbyte pages of each logical memory space in the physical memory.

Address Translation

To access a memory word, the processor translates the logical address of the word into a physical address and accesses the physical page, which contains the word. To facilitate the translation operation and indicate the location of logical addresses in physical memory, the processor uses map tables in the address translation facility.

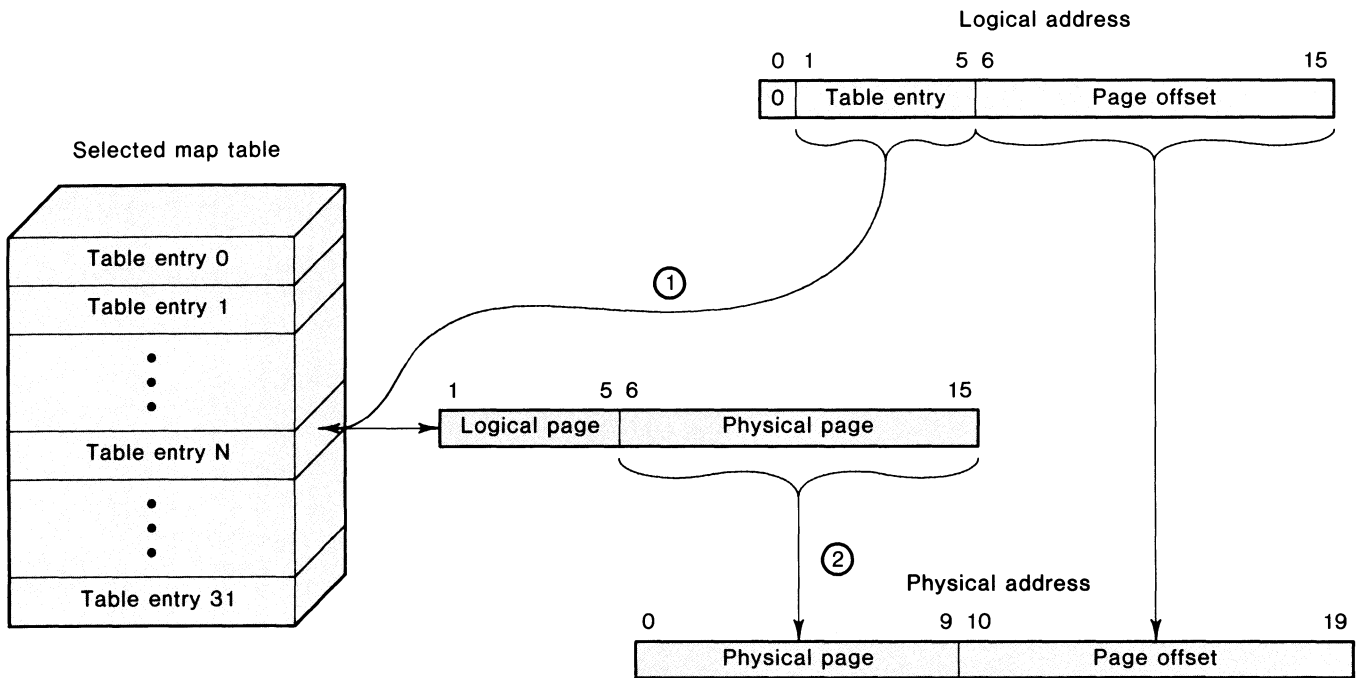
A map table contains one entry (map) for each 2 Kbyte logical page in the address space for a process. This entry indicates whether or not the process is allowed to access the page and gives information for logical-to-physical address translation.

When address translation is enabled, the processor uses the currently selected map table to translate the logical page address (most significant bits of the logical address) to a physical page address. The processor then combines the physical page address with the least significant 10 bits of the logical address to form the full physical address. Figure 7.1 illustrates this translation operation.

The translation facility contains separate map tables for user processes and data channel processes. The number of user and data channel map tables stored by the translation facility varies with the processor type. The stored map tables are referred to as user map table A, B, C,..., and data channel map table A, B, C,.... In computers with a burst multiplexor channel, the translation facility also includes a map table for processes using this I/O channel.

Each user requires a separate map table. While most translation facilities can store more than one user map table, the operating system's memory supervisor selects only one map table at a time for address translation. Similarly each data channel device requires a separate map table. However, unlike the case with user map tables, the interrupt service routine for the device controller requesting data channel service selects the data channel map table. Those controllers without an instruction capable of selecting map tables use data channel map A by default. The supervisor can enable or disable user or data channel mapping by manipulating bits in the user/DCH address translator's status register. Refer to the following "Status Registers" section for more information.

The basic formats for the user, data channel (DCH), and burst multiplexor channel (BMC) map table entries are diagrammed below. Refer to the specific assembly language programming manual for possible variations.



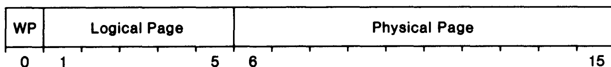
① The logical address to be translated has the format shown in the diagram. Bit 0 of the address (the indirection bit) is 0 because any indirection is resolved before address translation. Bits 1-5 of the address specify one of 32 map table entries (map regisers). Note that these bits are the logical address of a 2- Kbyte page (logical page). The processor uses the contents of these bits to access an entry in the user or data channel map table currently selected by the address translator's status register. This entry is labeled "Table entry N."

② Bits 6-15 of Table entry N, the physical page, become bits 0-9 of the physical page address. The page offset field specified in bits 6-15 of the logical address becomes bits 10-19 of the physical address. This is the 20-bit physical address translated from the original logical address. In processors that support less than 2 Mbytes of physical memory, the full 20-bit address is not used.

ID-00420

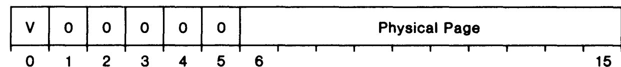
Figure 7.1 Address translation

User/DCH Map Table Entry Format



Bits	Name	Contents or Function
0	WP	User Write Protect flag for the physical page addressed by bits 6-15. (Write protection is only available for user processes. See the following "Protection" section.) If 0, allows user write access to the page. If 1, denies user write access to the page.
1-5	Logical Page	Logical address of a page in the address space of a process.
6-15	Physical Page	Physical address of the page with the logical address specified by bits 1-5.

BMC Map Table Entry Format



Bits	Name	Contents or Function
0	V	Validity (access) Protect flag for the page addressed by bits 1-15.
1-5	—	Must be 0.
6-15	Physical Page	Physical page address.

A special register, the *page 31 register*, allows any program, including the supervisor, to access any part of physical memory when user address translation is disabled (*unmapped mode*). In unmapped mode,

- Addresses in logical pages 0 to 30 are identical to the corresponding addresses in physical pages 0 to 30.
- Addresses in logical page 31 — addresses from 76000_8 to 77777_8 — are translated by a special map table entry stored in the page 31 register.

Protection

The address translation facilities also provide protection functions that enhance system integrity by preventing unauthorized access to certain parts of memory or I/O devices. Four types of protection are provided in systems with address translation facilities:

validity,
write,
indirection,
I/O.

Validity Protection

Validity protection protects a process' memory space from inadvertent access by another process, thereby preserving the integrity and privacy of the process' memory space. The supervisor should invalidate all logical pages unneeded by a process to ensure that mistaken references to unneeded paged do not result in unwanted access to another process' memory space. For example, if a user process needs only 12 pages (24 Kbyte), then logical pages 12 through 30 should be invalidated. To invalidate a page, the supervisor must set all the physical page bits to 1 and either the Write Protect flag (user/DCH map processes) or the Validity Protect flag (BMC processes) to 1. When address translation is disabled for a process, all logical pages are valid.

NOTE: *Most 16-bit real-time ECLIPSE processors do not provide validity protection for data channel processes.*

Validity protection is automatically enabled in systems with address translation facilities when user address translation is enabled. So, whenever the processor executes an instruction that references memory, it first checks the validity of the reference. If the reference is valid, the processor can access the page to read or write data, or to execute an instruction. If the reference is invalid, the processor aborts the executing memory reference instruction and services the protection violation. So the supervisor's responsibility for validity protection is limited to declaring the appropriate logical pages invalid.

Write Protection

Write protection permits *user processes* to read data from, but not write data to, protected memory pages. This provides the supervisor with a way of protecting the integrity of common areas of memory. To write-protect a page in a user's address space, the supervisor must set the Write Protect flag to 1 in the map table entry addressed by the logical page and enable write protection. When user address translation is enabled, the supervisor can enable or disable write protection at any time by manipulating the Write Protect Enable flag in the user/DCH address translation facility's status register. When user address translation is disabled, no write protection is provided.

Whenever the processor executes an instruction that references memory while user address translation and write protection are enabled, the processor checks for write protection before writing data into the addressed location. If the page containing the address is write protected, a protection fault occurs. The processor aborts the operation and services the protection violation.

Indirection Protection

An indirection loop occurs when the effective address calculation follows a chain of indirect addresses and never finds a direct address (one without bit 0 set to 0). Without indirection protection in such a case, the processor would never proceed to any further instructions and, thus, would effectively halt the system.

With indirection protection enabled, a specified number of indirect references (usually 16) causes a protection fault. The supervisor can enable or disable indirection protection at any time by manipulating the Indirection Protect Enable flag in the user/DCH address translation facility's status register.

I/O Protection

I/O protection protects the I/O devices in the system from unauthorized access. In many systems, all I/O operations are performed through operating system calls. In these systems, permitting individual users to execute I/O instructions is undesirable, since this would interfere with the operating system's handling of I/O devices. When a user process with I/O protection enabled attempts to execute an I/O instruction, a protection fault occurs. The processor does not execute the instruction, but services the protection fault. The supervisor can enable or disable I/O protection at any time by manipulating the I/O protection bit in the user/DCH address translator's status register.

Protection Fault Handling

Whenever a protection fault occurs, the processor inhibits access to the protected page.

In response to a *user* protection fault (validity, write, indirection, or I/O violation), the processor also

1. Disables user address translation.
2. For non-validity violations, sets the appropriate fault flag to 1 in the user/DCH address translator status register.
3. Pushes a fault return block onto the stack.
4. Checks for stack overflow. If 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 of the original fault. If no stack overflow occurs, the processor continues to service the original fault.

5. Jumps to the protection fault handler by an indirect jump through memory location 3, which should contain the protection fault handler routine address.

Responding to a *burst multiplexor channel* protection fault (validity violation), the processor sets the Validity Protect flag to 1 in both the BMC status register and the status register of the device that caused the fault.

The protection fault handler can determine the type of fault that occurred by reading the appropriate status register.

Address Translator Status

The user/DCH address translator contains a status register that controls the operation of the translator and provides information about protection violations. Various flags and fields in the status register

- Enable or disable user address translation;
- Enable or disable data channel address translation;
- Select the next user or data channel map table to be loaded;
- Select the next user map table to be used for address translation when user address translation is enabled;
- Enable or disable write protection, indirection protection, or I/O protection;
- Place the translator in LEF (Load Effective Address) mode;
- Indicate whether the last protection fault was a write, indirection, or I/O violation.

In computers with a burst multiplexor channel (BMC), information about the status of the BMC address translator is contained in the BMC status register. Various flags and fields in this register

- Specify the next BMC map table transfer operation (load or dump).
- Indicate that a BMC validity protection violation, address parity, or a data parity fault has occurred during data transfer between the BMC device and the BMC channel facility.

Tables 7.1 and 7.2 in the next section list the instructions for accessing these status registers.

Address Translator Instructions

Although they are functionally part of the memory system, the address translators are actually devices on the I/O bus and respond to the I/O instructions. The user/DCH address translator also responds to a special instruction (LMP). Tables 7.1 and 7.2 list the instructions for the user/DCH address translator and the BMC address translator, respectively.

Mnemonic	Name	Action
DIA MAP	*Read User/DCH Translator Status*	Returns translator status.
DIC MAP	*Page Check*	Returns the physical address and some characteristics of the logical page specified by the preceding DOC MAP instruction.
DOA MAP	*Load User/DCH Translator Status*	Specifies the operation of the address translator.
DOB MAP	*Translate Page 31*	When user address translation is disabled, selects the page 31 register as the map table entry for translating addresses in logical page 31.
DOC MAP	*Initiate Page Check*	Specifies the map table entry for a *Page Check* (DIC MAP) instruction without changing the other status of the translator.
IORST	*I/O Reset*	Disables user address translation and clears certain bits in the translator's status register.
LMP	*Load User/DCH Map Table*	Loads map table entries from memory into the translator.
NIOP MAP	*Translate Single Cycle*	Translates one memory reference using the last selected map table.
NIOP MAP	*Disable User Translation*	Disables user address translation.

Table 7.1 User/DCH address translator instructions

Mnemonic	Name	Action
DIC BMC	*Read BMC Status*	Returns translator and BMC facility status.
DOA BMC	*Specify Initial Address*	Specifies the least significant bits of the physical address of the first memory location to be accessed during the next map table transfer.
DOB BMC[1]	*Specify BMC Map Table Transfer*	Enables a map table transfer and specifies the most significant bits of the physical address of the first memory location to be accessed during the transfer.
DOB BMC[1]	*Select Initial BMC Map Entry*	Selects the first map table entry to be accessed during the next map table transfer.
DOC BMC	*Specify BMC Map Entry Count*	Specifies the number of map table entries to be loaded or dumped during the next map transfer.
IORST	*I/O Reset*	Clears certain flags in the BMC status register.

Table 7.2 BMC address translator instructions

[1] *Action depends on the contents of the specified accumulator.*

Error Checking and Correction

In addition to the protection features provided by the address translator, most 16-bit real-time ECLIPSE computers have an error checking and correction (ERCC) facility to enhance memory integrity.

During memory write operations, the ERCC facility generates and sends a multiple bit check code along with the word or double word written to memory. The number of bits in the check code depends on the organization of the particular memory system. In a memory system organized around single words (16 data bits), the check code contains 5 check bits; in a system organized around double words (32 data bits), the check code contains 7 bits. When enabled, the ERCC facility corrects all single-bit errors and detects multiple-bit errors in data read from memory. It also stores the address of the faulty data and a syndrome code which identifies the error. The memory supervisor can access this information for error logging and servicing.

ERCC Instructions

Although functionally part of the memory system, the ERCC facility is actually a device on the I/O bus and responds to I/O instructions. Table 7.3 lists the instructions for controlling the ERCC facility.

Mnemonic	Name	Action
DOA ERCC	*Enable ERCC*	Selects functions of the error checking and correction (ERCC) facility.
DIA ERCC	*Read Memory Fault Address*	Reads the least significant bits of the address of the erroneous data.
DIB ERCC	*Read Memory Fault Code and Address*	Reads the fault (syndrome) code and the most significant bits of the address of erroneous data.

Table 7.3 ERCC instructions

System Status and Special Functions

System management facilities are processor dependent. Most 16-bit real-time ECLIPSE processors have facilities that allow the operating system to perform the following special functions:

Halt program execution,
Transfer program control to a system call handler,
Read the virtual console data switch register.

Some 16-bit real-time ECLIPSE processors have a central processor identification register that stores information about the type of processor, the processor's microcode revision level, and the size of memory. For details on a processor's system management facilities, refer to the assembly language programming manual for the processor.

Chapter 8

Instruction Dictionary

ADC

Add Complement

ADC*[c][sh][#]* *acs,acd[,skip]*

1	ACS	ACD	1	0	0	SH	C	#	SKIP
0	1–2	3–4	5	6	7	8–9	10–11	12	13–15

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 ACS to the unsigned 16-bit number in ACD; and places the result in the shifter. The instruction then performs the specified shift operation and loads the result of the shift into ACD if the no-load bit is 0. If the skip condition is true, the next sequential word is skipped.

> **NOTE:** *If the number in ACS is less than the number in ACD, the instruction complements carry.*

Options

[c]

The processor determines the effect of carry (*c*) on the old value of carry before performing the operation (opcode). The following list gives the values of *c* and bits 10 and 11 and describes the operation.

Symbol *[c]*	Bits 10-11	Operation
Omitted	0 0	Leaves carry unchanged
Z	0 1	Initializes carry to 0
O	1 0	Initializes carry to 1
C	1 1	Complements carry

[sh]

The processor shifts carry and the 16 data bits after performing the instruction operation. The processor can shift the bits one bit position to the left or right or swap the two bytes. The following list gives the values of *sh* and bits 8 and 9 and describes the shift operation.

Symbol *[sh]*	Bits 8-9	Shift Operation
Omitted	0 0	Does not shift the result
L	0 1	Shifts left
R	1 0	Shifts right
S	1 1	Swaps 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 permits you 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 and bit 12 and describes the operation.

Symbol *[#]*	Bit 12	Operation
Omitted	0	Loads the result into ACD
#	1	Does not load the result; restores the carry bit

> **NOTE:** *You cannot combine the no-load option (#) with either the never skip or always skip option because the bit patterns for these combinations are used to define other types of instructions. This means that the ADC instruction cannot end in 1000_2 or 1001_2.*

[skip]

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

Symbol *[skip]*	Bits 13-15	Operation
Omitted	0 0 0	Never skips
SKP	0 0 1	Always skips
SZC	0 1 0	Skips if carry is 0
SNC	0 1 1	Skips if carry is not 0
SZR	1 0 0	Skips if the result is 0
SNR	1 0 1	Skips if the result is not 0
SEZ	1 1 0	Skips if either carry or the result is 0
SBN	1 1 1	Skips 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 an instruction that is 32 bits long.

Examples

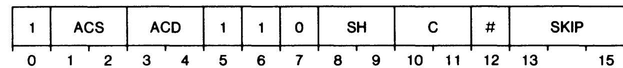
ADC 1,0

Adds the complement of AC1 to AC0. The parameters of the operation for two different sets of accumulator values follow.

Parameter	Before	After
AC0	000345_8	170243_8
AC1	010101_8	010101_8
Carry	0	0

Parameter	Before	After
AC0	010101_8	007533_8
AC1	000345_8	000345_8
Carry	0	1

ADD
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 ACS to the unsigned 16-bit number in ACD; and places the result in the shifter. The instruction then performs the specified shift operation and places the result of the shift in ACD if the no-load bit is 0. If the skip condition is true, the next sequential word is skipped.

> **NOTE:** *If the sum of the two numbers being added is greater than 65,535, the instruction complements carry.*

Options

[c]

The processor determines the effect of carry (*c*) on the old value of carry before performing the operation (opcode). The following list gives the values of *c* and bits 10 and 11 and describes the operation.

Symbol *[c]*	Bits 10-11	Operation
Omitted	0 0	Leaves carry unchanged
Z	0 1	Initializes carry to 0
O	1 0	Initializes carry to 1
C	1 1	Complements carry

[sh]

The processor shifts carry and the 16 data bits after performing the instruction operation. The processor can shift the bits one bit position to the left or right or swap the two bytes. The following list gives the values of *sh* and bits 8 and 9 and describes the shift operation.

Symbol *[sh]*	Bits 8-9	Shift Operation
Omitted	0 0	Does not shift the result
L	0 1	Shifts left
R	1 0	Shifts right
S	1 1	Swaps 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 permits you to test the result of the instruction operation without destroying the contents of the destination accumulator. The following list gives the values of the no-load option and bit 12 and describes the operation.

Symbol *[#]*	Bit 12	Operation
Omitted	0	Loads the result into ACD
#	1	Does not load the result; restores the carry bit

NOTE: *You cannot combine the no-load option (#) with either the never skip or always skip option because the bit patterns for these combinations are used to define other types of instructions. This means that the ADD instruction cannot end in* 1000_2 *or* 1001_2*.*

[skip]

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

Symbol *[skip]*	Bits 13-15	Operation
Omitted	0 0 0	Never skips
SKP	0 0 1	Always skips
SZC	0 1 0	Skips if carry is 0
SNC	0 1 1	Skips if carry is not 0
SZR	1 0 0	Skips if the result is 0
SNR	1 0 1	Skips if the result is not 0
SEZ	1 1 0	Skips if either carry or the result is 0
SBN	1 1 1	Skips 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 an instruction that is 32 bits long.

Examples

```
ADD     1,0
```

Adds AC1 to AC0. The parameters of the operation for three different sets of accumulator values follow.

Parameter	Before	After
AC0 AC1	000345_8 010101_8	010446_8 010101_8
Carry	0	0

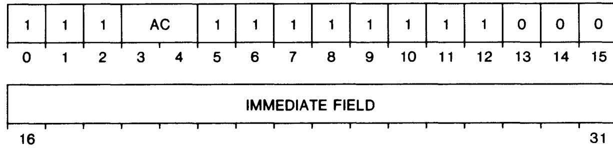
Parameter	Before	After
AC0	077777_8	100000_8
AC1	000001_8	000001_8
Carry	0	0

Parameter	Before	After
AC0	177777_8	000000_8
AC1	000001_8	000001_8
Carry	0	1

ADDI

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 the specified accumulator, placing the result in the same accumulator. Carry remains unchanged.

Examples

ADDI 303,1

Adds 303_8 to AC1. The parameters of the operation follow.

Parameter	Before	After
AC1	000345_8	000650_8

ADDI −1,1

Adds 177777_8 to AC1. The parameters of the operation follow.

Parameter	Before	After
AC1	177777_8	177776_8

ADDI 77777,1

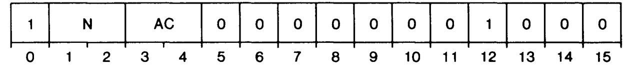
Adds 77777_8 to AC1. The parameters of the operation follow.

Parameter	Before	After
AC1	177777_8	07777_8

ADI

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 in the specified accumulator, placing the result in the same accumulator. The carry bit remains unchanged.

> **NOTE:** *DGC assemblers take the coded value* n *and subtract 1 from it before placing it in the immediate field. Therefore, you should code the exact value that is to be added.*

Examples

ADI 3,2

Adds 3 to AC2. The parameters of the operation follow.

Parameter	Before	After
AC2	000050_8	000053_8

ADI 4,2

Adds 4 to AC2. The parameters of the operation follow.

Parameter	Before	After
AC2	177775_8	000001_8

ANC

AND with Complemented Source

ANC *acs,acd*

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

Logically 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 ACS and the contents of ACD and places the result in ACD. The instruction sets a bit position in the result to 1 if the corresponding bits in ACS and ACD contain 0 and 1, respectively. The contents of ACS remain unchanged.

Example

ANC 0,1

ANDs the complement of AC0 with AC1. The parameters of the operation follow.

Parameter	Before	After
AC0	177775_8	177775_8
AC1	000112_8	000002_8

AND

AND

AND*[c][sh][#]* *acs,acd[,skip]*

1	ACS	ACD	1	1	1	SH	C	#	SKIP
0	1–2	3–4	5	6	7	8–9	10–11	12	13–15

Logically ANDs of the contents of two accumulators.

Initializes carry to the specified value. Places the logical AND of the contents of ACS and ACD in the shifter. Each bit placed in the shifter is 1 only if the corresponding bits in both ACS and ACD are 1; otherwise, the resulting bit is 0. The instruction then performs the specified shift operation and places the result in ACD if the no-load bit is 0. If the skip condition is true, the next sequential word is skipped.

Options

[c]

The processor determines the effect of carry (*c*) on the old value of carry before performing the operation (opcode). The following list gives the values of *c* and bits 10 and 11 and describes the operation.

Symbol *[c]*	Bits 10-11	Operation
Omitted	0 0	Leaves carry unchanged
Z	0 1	Initializes carry to 0
O	1 0	Initializes carry to 1
C	1 1	Complements carry

[sh]

The processor shifts carry and the 16 data bits after performing the instruction operation. The processor can shift the bits one bit position to the left or right or swap the two bytes. The following list gives the values of *sh* and bits 8 and 9 and describes the shift operation.

Symbol *[sh]*	Bits 8-9	Shift Operation
Omitted	0 0	Does not shift the result
L	0 1	Shifts left
R	1 0	Shifts right
S	1 1	Swaps 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 permits you 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 and bit 12 and describes the operation.

Symbol *[#]*	**Bit 12**	**Operation**
Omitted	0	Loads the result into ACD
#	1	Does not load the result; restores the carry bit

NOTE: *You cannot combine the no-load option (#) with either the never skip or always skip option because the bit patterns for these combinations are used to define other types of instructions. This means that AND instruction cannot end in* 1000_2 *or* 1001_2.

[skip]

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

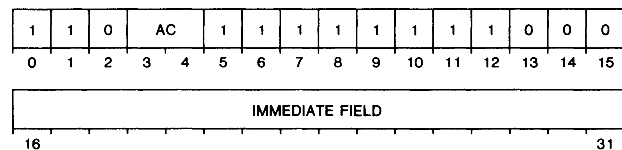
Symbol *[skip]*	**Bits 13-15**	**Operation**
Omitted	0 0 0	Never skips
SKP	0 0 1	Always skips
SZC	0 1 0	Skips if carry is 0
SNC	0 1 1	Skips if carry is not 0
SZR	1 0 0	Skips if the result is 0
SNR	1 0 1	Skips if the result is not 0
SEZ	1 1 0	Skips if either carry or the result is 0
SBN	1 1 1	Skips if 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 an instruction that is 32 bits long.

ANDI

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 the specified accumulator in the specified accumulator. Carry is unchanged.

BAM

Block Add and Move

1	0	0	1	0	1	1	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 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 AC0 to it. If the addition produces a result that is greater than 32,768, no indication is given.

Bits 1–15 of AC2 contain the address of the source location. Bits 1–15 of AC3 contain the address of the destination location. The address in bits 1–15 of AC2 or AC3 is an indirect address if bit 0 of that accumulator is 1. If the address is an indirect address, the instruction follows the indirection chain before placing the resultant effective address in the accumulator.

The unsigned 16-bit number in 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.

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 AC0 remain unchanged.

Words are moved in consecutive ascending order according to their addresses. The next address after 77777_8 is 0 for both fields. The fields can overlap in any way.

NOTES: *Because of its potentially long execution time, this instruction 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 is 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.*

Example

BAM

Moves 20_8 words from locations 77770_8+ to locations 34450_8+, adding 32_8 to each word. The parameters of the operation follow.

Parameter	Before	After
AC0	000031_8	000031_8
AC1	000020_8	000000_8
AC2	077770_8	000011_8
AC3	034450_8	034471_8

BLM

Block Move

1	0	1	1	0	1	1	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 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.

> **NOTE:** *The* Block Move *instruction is interruptible in the same manner as the* Block Add and Move *instruction.*

Example

BLM

Moves a block of 20_8 words from locations 77770_8+ to locations 34450_8+. The parameters of the operation follow.

Parameter	Before	After
AC1	000020_8	000000_8
AC2	077770_8	000010_8
AC3	034450_8	034470_8

BTO

Set Bit to One

BTO *acs,acd*

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

Sets the specified bit to 1.

Forms a 32-bit bit pointer from the contents of both ACS and ACD. ACS contains the most significant 16 bits and ACD the least significant 16 bits of the bit pointer. If ACS and ACD are specified as the same accumulator, the instruction treats the accumulator contents as the least significant 16 bits of the bit pointer and assumes the most significant 16 bits are 0.

The instruction then sets the addressed bit in memory to 1, leaving the contents of ACS and ACD unchanged.

The 15-bit effective address generated by this instruction is an indirect address if bit 0 of the bit pointer is 1.

Example

BTO 1,0

Sets the bit addressed by the bit pointer in AC1 and AC0 to 1. The parameters for setting bit 7 at location 23456_8 follow.

Parameter	Before	After
Location 23456_8	010103_8	010503_8
ACO	023450_8	023450_8
AC1	000147_8	000147_8

BTZ

Set Bit to Zero

BTZ *acs,acd*

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

Sets the addressed bit to 0.

Forms a 32-bit bit pointer from the contents of both ACS and ACD. ACS contains the most significant 16 bits and ACD contains the least significant 16 bits of the bit pointer. If ACS and ACD are specified as the same accumulator, the instruction treats the accumulator contents as the least significant 16 bits of the bit pointer and assumes the most significant 16 bits are 0.

The instruction then sets the addressed bit in memory to 0, leaving the contents of ACS and ACD unchanged.

The 15-bit effective address generated by this instruction is an indirect address if bit 0 of the bit pointer is 1.

Example

```
BTZ     1,0
```

Sets the bit addressed by the bit pointer in AC1 and AC0 to 0. The parameters for setting bit 7 at location 23456_8 follow.

Parameter	Before	After
Location 23456_8	010503_8	010103_8
AC0	023450_8	023450_8
AC1	000147_8	000147_8

CLM

Compare to Limits

CLM *acs,acd*

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

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 signed 16-bit two's complement integer in ACS to two signed 16-bit two's complement limit values, L and H. If the number in 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 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 1–15 of ACD. Bit 0 of ACD is ignored. The limit value H is contained in the word following L.

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.

Example

```
CLM     1,0
```

Compares the number in AC1 to the low limit in the location addressed by AC0 and the high limit addressed by AC0+1. Because the number is between the limit values, the processor skips the next sequential word after the word containing the CLM instruction. The parameters of the operation follow.

Parameter	Before	After
AC0	001234_8	001234_8
AC1	000331_8	000331_8
Location 001234_8	000017_8	000017_8

CMP

Character Compare

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

Under control of the four accumulators, compares two strings of bytes and returns a code in AC1 reflecting the results of the comparison.

The instruction compares the strings one byte at a time. Each byte is treated as an unsigned 8-bit binary quantity in the range 0–255_{10}. If two bytes are not equal, the string whose byte has the smaller numerical value is, by definition, the *lower valued* string. Both strings remain unchanged.

The four accumulators define the operation as follows:

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, AC1 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.

The strings can overlap in any way. Overlap will not affect the results of the comparison.

After completion of the character compare operation, the contents of the accumulators follow.

AC0 contains the number of bytes left to compare in string 2.

AC1 Return code as shown in the list below.

AC2 contains 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).

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).

Code	Comparison Result
−1	string 1 < string 2
0	string 1 = string 2
+1	string 1 > string 2

NOTE: *If AC0 and AC1 both contain 0 (both string 1 and string 2 have length zero), the instruction compares no bytes and returns 0 in AC1. If the two strings are of unequal length, the instruction pads the shorter string with space characters* $<040_8>$ *and continues the comparison.*

Example

CMP

Compares the byte string at locations 000345_8 through 000400_8 to the byte string starting at location 011123_8, from lowest to highest location. The parameters of the operation follow.

Parameter	Before	After
ACO	000066_8	000000_8
AC1	000066_8	Return code
AC2	000713_8	001001_8
AC3	022247_8	022335_8

CMT

Character Move Until True

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

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 0, 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 define the operation as follows:

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.

When the source and destination addresses are the same (AC2 equals AC3), no bytes are moved, and the string is scanned for the delimiter. Any other type of overlap may produce unusual side effects.

After completion of the character move operation, the accumulators are as follows:

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.

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.*

Example

CMT

Checks the 42_8 byte string beginning at location 001132_8 for a delimiter, beginning at the string's lowest byte. The parameters of the operation follow.

Parameter	Before	After
AC0	010203_8	010203_8
AC1	000042_8	000000_8 (no delimiter found)
AC2	002265_8	002337_8 (no delimiter found)
AC3	002265_8	002337_8 (no delimiter found)

CMV
Character Move

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

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 the source and destination strings.

The instruction copies the source string to the destination field, one byte at a time. The four accumulators define the operation as follows:

AC0 specifies the length and direction of processing for the destination field. If the field is to be processed from its lowest memory location to the highest, AC0 contains the unsigned value of the number of bytes in the destination field. If the field is to be processed from its highest memory location to the lowest, AC0 contains the two's complement of the number of bytes in the destination field.

AC1 specifies the length and direction of processing for the source string. If the string is to be processed from its lowest memory location to the highest, AC1 contains the unsigned value of the number of bytes in the source string. If the field is to be processed from its highest memory location to the lowest, AC1 contains the two's complement of the number of bytes in the source string.

AC2 contains a byte pointer to the first byte to be written in the destination field. When the field is written in ascending order, AC2 points to the lowest byte. When the field is written in descending order, AC2 points to the highest byte.

AC3 contains a byte pointer to the first byte copied in the source string. When the field is copied in ascending order, AC3 points to the lowest byte. When the field is copied in descending order, AC3 points to the highest byte.

The fields can overlap in any way. However, the instruction moves bytes one at a time, so certain types of overlap may produce unusual side effects.

After completion of the character move operation, the accumulators are as follows:

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.

AC3 contains a byte pointer to the byte following the last byte fetched from the source field.

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* <040_8>.

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 every 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>.

Example

CMV

Moves a string of bytes starting at location 003321_8 to the location starting at 001111_8. The string being moved is 30_8 bytes long and the destination string is 40_8 bytes long, and both are to be processed in ascending order. Because the source string is shorter than the destination string, the last 10_8 bytes of the destination string are filled with space characters and the carry is set to 0.

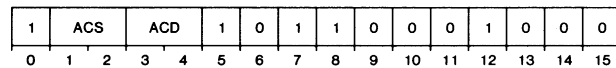
The parameters of the operation follow.

Parameter	Before	After
AC0	000040_8	000000_8
AC1	000030_8	000000_8
AC2	002223_8	002263_8
AC3	006643_8	006673_8
Carry	Unknown	0

COB

Count Bits

COB *acs,acd*



Counts the number of ones in an accumulator.

Adds a number equal to the number of ones in ACS to the signed 16-bit two's complement in ACD. The contents of ACS remain unchanged.

> **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.*

Example

COB 1,0

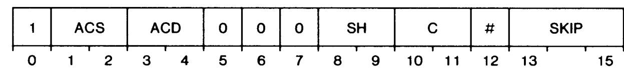
Adds a number equal to the number of ones in AC1 to the signed number in AC0. The parameters of the operation follow.

Parameter	Before	After
AC0	123450_8	123453_8
AC1	000103_8	000103_8

COM

Complement

COM[*c*][*sh*][*#*] *acs,acd[,skip]*



Logically complements the contents of an accumulator.

Initializes carry to the specified value, forms the logical complement of the number in ACS, and performs the specified shift operation. The instruction then places the result in ACD if the no-load bit is 0. If the skip condition is true, the next sequential word is skipped.

Options

[c]

The processor determines the effect of carry (*c*) on the old value of carry before performing the operation (opcode). The following list gives the values of *c* and bits 10 and 11 and describes the operation.

Symbol *[c]*	Bits 10-11	Operation
Omitted	0 0	Leaves carry unchanged
Z	0 1	Initializes carry to 0
O	1 0	Initializes carry to 1
C	1 1	Complements carry

[sh]

The processor shifts carry and the 16 data bits after performing the instruction operation. The processor can shift the bits one bit position to the right or left or swap the two bytes. The following list gives the values of *sh* and bits 8 and 9 and describes the shift operation.

Symbol *[sh]*	Bits 8-9	Shift Operation
Omitted	0 0	Does not shift the result
L	0 1	Shifts left
R	1 0	Shifts right
S	1 1	Swaps 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 permits you 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 and bit 12 and describes the operation.

Symbol *[#]*	Bit 12	Operation
Omitted	0	Loads the result into ADC
#	1	Does not load the result; restores the carry bit

NOTE: *You cannot combine the no-load option (#) with either the never skip or always skip option because the bit patterns of these combinations are used to define other types of instructions. This means that the COM instruction cannot end in* 1000_2 *or* 1001_2.

[skip]

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

Symbol *[skip]*	Bits 13-15	Operation
Omitted	0 0 0	Never skips
SKP	0 0 1	Always skips
SZC	0 1 0	Skips if carry is 0
SNC	0 1 1	Skips if carry is not 0
SZR	1 0 0	Skips if the result is 0
SNR	1 0 1	Skips if the result is not 0
SEZ	1 1 0	Skips if either carry or the result is 0
SBN	1 1 1	Skips 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 an instruction that is 32 bits long.

CTR

Character Translate

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

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 with a second translated string.

The instruction operates in two modes: translate and move, and translate and compare.

Translate and Move Mode

Translate and move mode is specified by a 1 in bit 0 of AC1. In this 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.

The accumulators define the operation as follows:

AC0 specifies the address, either direct or indirect, of a word that 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. It contains the two's complement of the number of bytes in the strings. Both strings are processed from lowest memory address to highest.

NOTE: *Because bit 0 of AC1 contains a 1, the maximum number of bytes to be moved is* $32{,}767_{10}$.

AC2 contains a byte pointer to the first byte in string 2.

AC3 contains a byte pointer to the first byte in string 1.

Upon completion of a translate and move operation, the accumulators are as follows:

AC0 contains the address of the word that contains the byte pointer to the translation table, and AC1 contains 0.

AC2 contains a byte pointer to the byte following string 2.

AC3 contains a byte pointer to the byte following string 1.

The fields can 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.

Translate and Compare Mode

Translate and compare mode is specified by a 0 in bit 0 of AC1. In this mode, the instruction translates each byte in string 1 and string 2 as described above and compares the translated values. Each translated byte is treated as an unsigned 8-bit binary quantity in the range $0–255_{10}$. If two translated bytes are not equal, the string whose byte has the smaller numerical value is, by definition, the *lower valued* string. Both strings remain unchanged.

The accumulators define the operation as follows:

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. It 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 byte pointer to the first byte in string 2.

AC3 contains a byte pointer to the first byte in string 1.

After completion of a translate and compare operation, the accumulators are as follows:

AC0 contains the address of the word that 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 were 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 were found) or the byte following string 1 (if the strings were identical).

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 lengths of both string 1 and string 2 are 0, the compare option returns a 0 in AC1.

The fields can 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.

Example

CTR

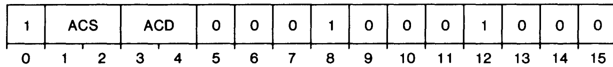
In translate and move mode: translates a 50-byte string starting at location 010321_8 using a predefined translation table and returns the string to its original location. The source string will be overwritten and its locations filled with the translated string. The parameters for the operation follow.

Parameter	Before	After
Location 002123_8	Byte pointer to translation table	Byte pointer to translation table
AC0	002123_8	002123_8
AC1	177716_8	000000_8
AC2	020643_8	020725_8
AC3	020643_8	020725_8

DAD

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 12–15 of ACS to the unsigned decimal digit contained in bits 12–15 of ACD. Carry is added to this result. The instruction then places the least significant decimal digit of the final result in bits 12–15 of ACD and the decimal carry in carry. The contents of ACS and bits 0–11 of ACD remain unchanged.

> **NOTE:** *No validation of the input digits is performed. Therefore, if bits 12–15 of either ACS or ACD contain a number greater than 9, the results will be unpredictable.*

Example

DAD 2,3

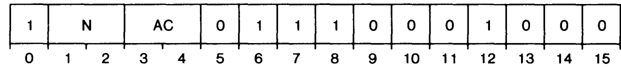
Adds AC2 (which contains 9_{10}) to AC3 (which contains 7_8). The parameters of the operation follow.

Parameter	Before	After
AC2, bits 12-15	$9_{10}(1001_2)$	$9_{10}(1001_2)$
AC3, bits 12-15	$7_{10}(0111_2)$	$6_{10}(0110_2)$
Carry	0	1

DHXL

Double Hex Shift Left

DHXL *n,ac*



Shifts the 32-bit contents of two accumulators left 1 to 4 hex digits.

Shifts the 32-bit number contained in AC and 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 zeros.

> **NOTES:** *If AC is specified as AC3, then AC+1 is AC0.*
>
> *DGC assemblers take the coded value of* n *and subtract 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 zeros.*

Example

DHXL 1,0

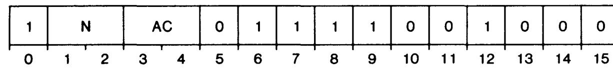
Shifts the 32-bit number contained in AC0 (most significant bits) and AC1 (least significant bits) left one hex digit. The parameters for the operation are as follows:

Parameter	Before	After
AC0	001160_8	023405_8
AC1	050010_8	000200_8

DHXR

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 AC and 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 zeros.

> **NOTES:** *If AC is specified as AC3, then AC+1 is AC0.*
>
> *DGC assemblers take the coded value of* n *and subtract 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 zeros.*

Example

DHXR 2,0

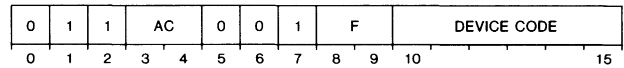
Divides the number in AC0 and AC1 by 256_{10}, that is, shifts the number right two hex digits. The parameters of the operation follow.

Parameter	Before	After
AC0	001160_8	000002_8
AC1	050010_8	070120_8

DIA

Data In A

DIA *[f] ac,device*



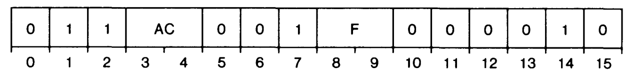
Transfers data from the A buffer of an I/O device to an accumulator.

The contents of the A input buffer in the specified device are placed in the specified accumulator. After the data transfer, the Busy and Done flags are set according to the function specified by *f*.

DIA ERCC

Read Memory Fault Address

DIA*[f]* *ac*,**ERCC**



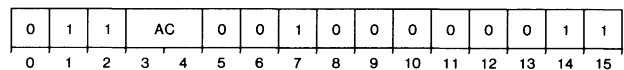
In computers with error checking and correction (ERCC), reads the least significant bits of the memory fault address.

For more information, refer to the assembly language programming manual for the specific computer.

DIA MAP

Read User/DCH Translator (MAP) Status

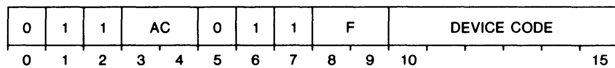
DIA *ac*,**MAP**



In computers with address translation, supplies the status of the user/DCH address translator.

For more information, refer to the assembly language programming manual for the specific computer.

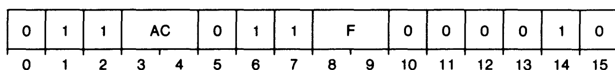
DIB
Data In B
DIB *[f] ac,device*



Transfers data from the B buffer of an I/O device to an accumulator.

Places the contents of the B input buffer in the specified device in the specified accumulator. After the data transfer, sets the Busy and Done flags according to the function specified by *f*.

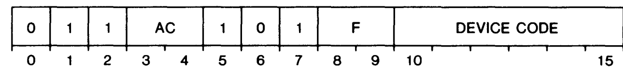
DIB ERCC
Read Memory Fault Code and Address
DIB*[f]* *ac*,ERCC



In computers with error checking and correction (ERCC), reads the fault code and the most significant bits of the memory fault address.

For more information, refer to the assembly language programming manual for the specific computer.

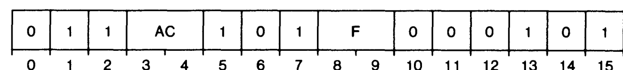
DIC
Data In C
DIC *[f] ac,device*



Transfers data from the C buffer of an I/O device to an accumulator.

Places the contents of the C input buffer in the specified device in the specified accumulator. After the data transfer, sets the Busy and Done flags according to the function specified by *f*.

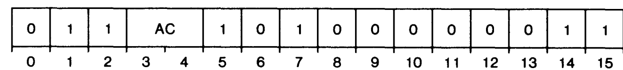
DIC BMC
Read BMC Status
DIC*[f]* *ac*,BMC



In computers with a burst multiplexor channel (BMC), returns the status of the BMC address translator and the BMC facility.

For more information, refer to the assembly language programming manual for the specific computer.

DIC MAP
Page Check
DIC *ac*,MAP



In computers with address translation, returns the physical address and the state of the Write Protect flag for the logical page specified by the preceding *Initiate Page Check* instruction (DOC *ac*,MAP).

For more information, refer to the assembly language programming manual for the specific computer.

DIV

Unsigned Divide

1	1	0	1	0	1	1	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

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

> **NOTE:** *Before the divide operation takes place, the number in AC0 is compared with the number in AC2. If the contents of AC0 are greater than or equal to the contents of AC2, an overflow condition is indicated. Carry is set to 1, and the operation is terminated. All operands remain unchanged.*

Examples

DIV

Four different DIV examples are given below. The first example divides 6 by 2. The parameters of the operation follow.

Parameter	Before	After
AC0	000000_8	000000_8
AC1	000006_8	000003_8
AC2	000002_8	000002_8
Carry	0	0

Divides 7777600001_8 by 077777_8. The parameters of the operation follow.

Parameter	Before	After
AC0	037777_8	000000_8
AC1	000001_8	077777_8
AC2	077777_8	000002_8
Carry	0	0

Divides 3777740001_8 by 177777_8. An overflow results. The parameters of the operation follow.

Parameter	Before	After
AC0	177776_8	177776_8
AC1	000001_8	000001_8
AC2	077777_8	077777_8
Carry	0	1

Divides 7 by 2. A remainder of 1 results. The parameters of the operation follow.

Parameter	Before	After
AC0	000000_8	000001_8
AC1	000007_8	000003_8
AC2	000002_8	000002_8
Carry	0	0

DIVS

Signed Divide

1	1	0	1	1	1	1	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

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

Divides the signed 32-bit two's complement contained in AC0 and AC1 by the signed 16-bit two's complement in AC2. The quotient and remainder are signed 16-bit numbers and occupy 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; however, a 0 quotient or a 0 remainder is always positive. Carry is set to 0. The contents of AC2 remain unchanged.

> **NOTE:** *If the magnitude of the quotient is such that it will not fit into AC1, an overflow condition is indicated. Carry is set to 1, and the operation is terminated. The contents of AC0 and AC1 are unpredictable.*

Examples

DIVS

Four different DIVS examples are given below. The first example divides 6 by 2. The parameters of the operation follow.

Parameter	Before	After
AC0	000000_8	000000_8
AC1	000006_8	000003_8
AC2	000002_8	000002_8

Divides 7777600001_8 by 077777_8. The parameters of the operation follow.

Parameter	Before	After
AC0	037777_8	000000_8
AC1	000001_8	077777_8
AC2	077777_8	077777_8

Divides 1 by −1. A remainder of −1 results. The parameters of the operation follow.

Parameter	Before	After
AC0	000000_8	000000_8
AC1	000001_8	177777_8
AC2	177776_8	177776_8

Divides −7 by −2. A remainder of −1 results. The parameters of the operation follow.

Parameter	Before	After
AC0	177777_8	177777_8
AC1	177777_8	000003_8
AC2	177776_8	177776_8

DIVX

Sign Extend and Divide

1	0	1	1	1	1	1	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

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 AC1 into AC0 by placing a copy of bit 0 of AC1 in each bit of AC0. After extending the sign, the instruction performs a DIVS (*Signed Divide*) operation.

DLSH

Double Logical Shift

DLSH *acs,acd*

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

Logically shifts the 32-bit contents of two accumulators left or right, depending on the contents of a third accumulator.

Shifts the 32-bit number contained in ACD and ACD+1 left or right, depending on the number contained in bits 8–15 of ACS. The signed 8-bit two's complement contained in bits 8–15 of ACS determines the direction of the shift and the number of bits to be shifted. If the number in bits 8–15 of ACS is positive, the shift is to the left; if the number in bits 8–15 of ACS is negative, the shift is to the right. If the number in bits 8–15 of ACS is zero, no shifting is performed. Bits 0–7 of ACS are ignored.

AC3+1 is AC0. The number of bits shifted is equal to the magnitude of the number in bits 8–15 of ACS. Bits shifted out are lost, and the vacated bit positions are filled with zeros. The contents of ACS remain unchanged.

> **NOTE:** *If the magnitude of the number in bits 8–15 of ACS is greater than* 31_{10}, *all bits of ACD are set to 0. Carry and ACS remain unchanged.*

Example

DLSH 0,1

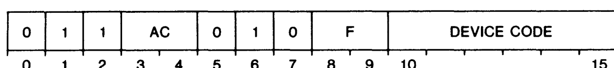
Two different DLSH 0, 1 examples are given below. This first example shifts left one bit. The parameters of the operation follow.

Parameter	Before	After
AC0	000001_8	000001_8
AC1	012345_8	024712_8
AC2	054321_8	130642_8

Shifts right one bit. The parameters of the operation follow.

Parameter	Before	After
AC0	000377_8	000377_8
AC1	024712_8	012345_8
AC2	130642_8	054321_8

DOA
Data Out A
DOA *device*

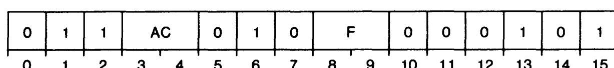


Transfers data from an accumulator to the A buffer of an I/O device.

Places the contents of the bits 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.

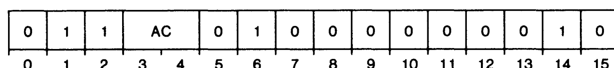
DOA BMC
Specify Initial Address; Specify Low-Order Address
DOA*[f]* *ac*,**BMC**



In computers with a burst multiplexor channel, specifies the least significant address bits of the first memory location of supply or receives a word during the next map transfer (load or dump) operation.

For details, refer to the assembly language programming manual for the specific computer.

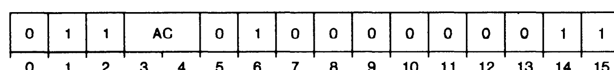
DOA ERCC
Enable ERCC
DOA*[f]* *ac*,**ERCC**



In computers with error checking and correction (ERCC), selects functions of the ERCC facility.

For more information, refer to the assembly language programming manual for the specific computer.

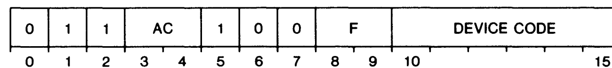
DOA MAP
Load User/DCH Translator (MAP) Status
DOA *ac*,**MAP**



In computers with address translation, specifies the operation of the user/data channel address translator.

For more information, refer to the assembly language programming manual for the specific computer.

DOB
Data Out B
DOB *device*



Transfers data from an accumulator to the B buffer of an I/O device.

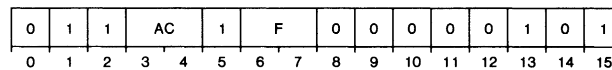
Places the contents of the specified accumulator in the B output buffer of the specified device. After the data transfer, sets the Busy and Done flags according to the function specified by *f*. The contents of the specified accumulator remain unchanged.

The number of data bits moved depends upon the size of the buffer and the mode of operation of the device.

DOB BMC
Specify BMC Map Table Transfer; Specify Operation and High-Order Address

or

Select Initial BMC Map Entry; Specify Initial Map Register
DOB*[f]* *ac*,**BMC**

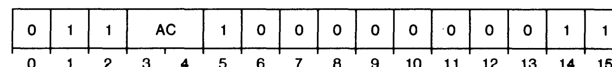


In computers with a burst multiplexor channel (BMC), this instruction performs one of the following two functions:

- specifies the BMC map table transfer operation (load or dump) and the most significant address bits of the first memory location to supply or receive a word during the operation
- selects the first BMC map table entry to receive or supply a word during the next map table transfer (load or dump) operation

For more information, refer to the assembly language programming manual for the specific computer.

DOB MAP
Translate Page 31; Map Supervisor Page 31
DOB *ac*,**MAP**



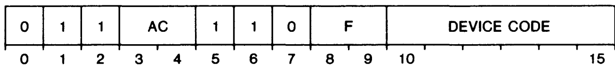
In computers with address translation, supplies the physical address for the translation of user memory references to logical page 31 when user mapping is disabled.

For more information, refer to the assembly language programming manual for the specific computer.

DOC

Data Out C

DOC *device*



Transfers data from an accumulator to the C buffer of an I/O device.

Places the contents 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.

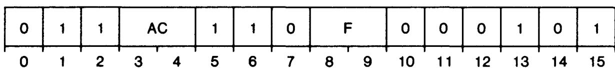
The number of data bits moved depends upon the size of the buffer and the mode of operation of the device.

> **NOTE:** *DGC assemblers reserve the DOC 0,CPU instruction* (Halt) *for stopping the processor.*

DOC BMC

Specify BMC Map Entry Count

DOC*[f]* *ac*,**BMC**



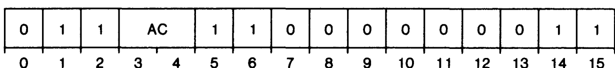
In computers with a burst multiplexor channel (BMC), specifies the number of BMC map table entries to be loaded or dumped during the next map table transfer operation.

For more information, refer to the assembly language programming manual for the specific computer.

DOC MAP

Initiate Page Check

DOC *ac*,**MAP**



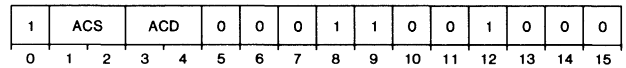
In computers with address translation, selects a map table and specifies a table entry.

For more information, refer to the assembly language programming manual for the specific computer.

DSB

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 bits 12–15 of ACS from the unsigned decimal digit contained in bits 12–15 of ACD. Subtracts the complement of carry from this result. Places the least significant decimal digit of the final result in bits 12–15 of ACD 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 0–11 of ACD remain unchanged.

Example

```
DSB    3,2
```

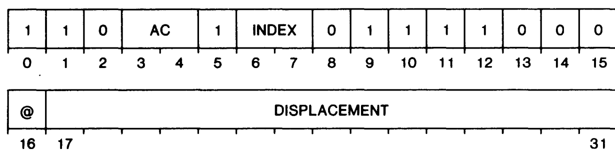
Subtracts AC3 (which contains 7_{10}) from AC2 (which contains 9_{10}). The parameters of the operation follow.

Parameter	Before	After
AC2, bits 12-15	$9_{10}(1001_2)$	$9_{10}(1001_2)$
AC3, bits 12-15	$7_{10}(0111_2)$	$2_{10}(0010_2)$
Carry	0	1

DSPA

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 8.1, the dispatch table consists of a table of addresses. Immediately before the table are two signed 16-bit two's complement limit words, L and H. The last word of the table is in location $E+H-L$.

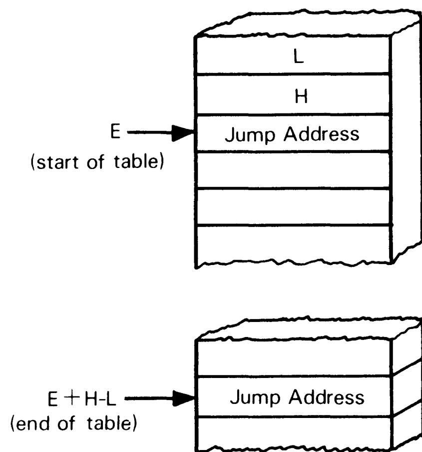


Figure 8.1 Dispatch table

Compares the signed two's complement contained in 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 the specified accumulator is greater than or equal to L and less than or equal to H, the instruction fetches the word at location $E-L+number$. If the fetched word is equal to 177777_8, sequential operation continues with the instruction immediately after the *Dispatch* instruction. If the fetched word is not equal to 177777_8, the instruction treats this word as the 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 and sequential operation continues with the word addressed by the updated value of the program counter.

Figure 8.2 shows the dispatch operation.

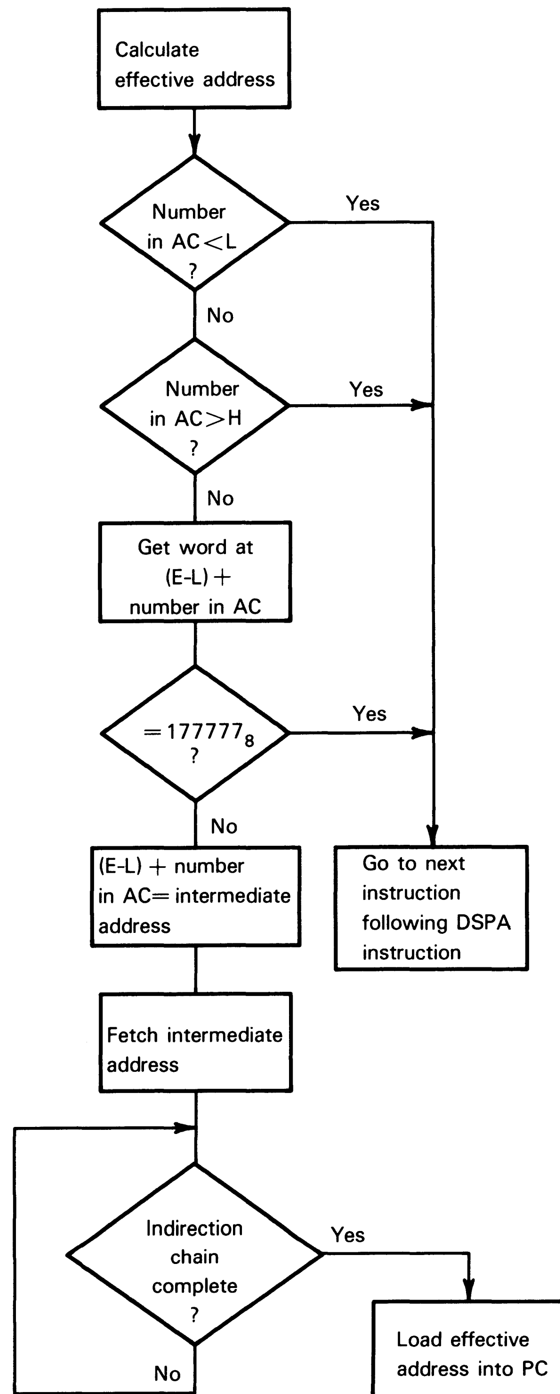
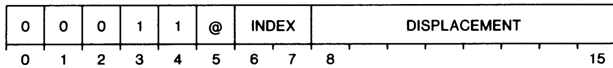


Figure 8.2 Dispatch operation

DSZ

Decrement and Skip if Zero

DSZ *[@]displacement[,index]*



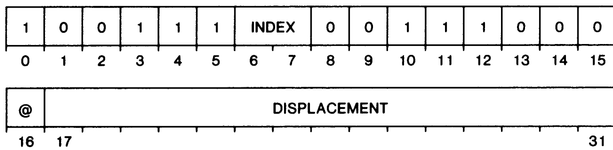
Decrements the addressed word; skips if the decremented value is 0.

Computes the effective address (*E*). Decrements by one the word addressed by *E* and writes the result into that location. If the updated value of the location is 0, the instruction skips the next sequential word.

EDSZ

Extended Decrement and Skip if Zero

EDSZ *[@]displacement[,index]*



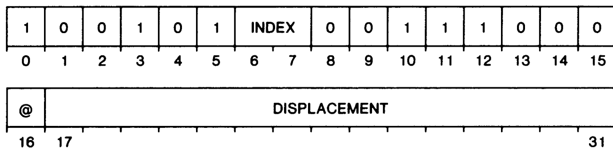
Decrements the addressed word; skips if the decremented value is 0.

Computes the effective address (*E*). Decrements by one the contents of the location addressed by *E* and writes the result into memory at the same address. If the updated value of the word is 0, the instruction skips the next sequential word.

EISZ

Extended Increment and Skip if Zero

EISZ *[@]displacement[,index]*



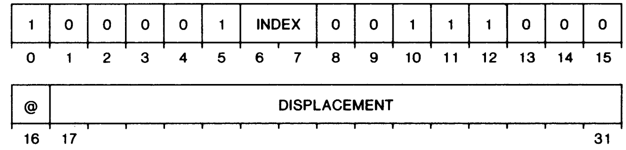
Increments the addressed word; skips if the incremented value is 0.

Computes the effective address (*E*). Increments by one the contents of the location specified by *E* and writes the result into memory at the same address. If the updated value of the location is 0, the instruction skips the next sequential word.

EJMP

Extended Jump

EJMP *[@]displacement[,index]*



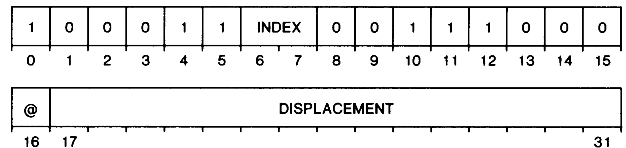
Loads an effective address into the program counter.

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.

EJSR

Extended Jump to Subroutine

EJSR *[@]displacement[,index]*



Increments and stores the value of the program counter in an accumulator, then places a new address in the program counter.

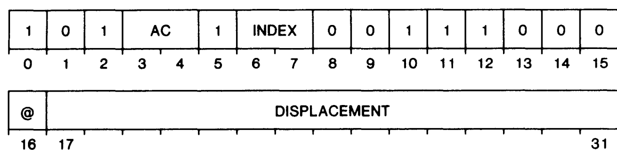
Computes the effective address (*E*). Places the address of the next sequential instruction (the instruction following the EJSR instruction) in AC3 and *E* in the program counter. Sequential operation continues with the word addressed by the updated value of the program counter.

> **NOTE:** *The instruction computes* E *before it places the incremented program counter in AC3.*

ELDA

Extended Load Accumulator

ELDA *ac[@]displacement[,index]*



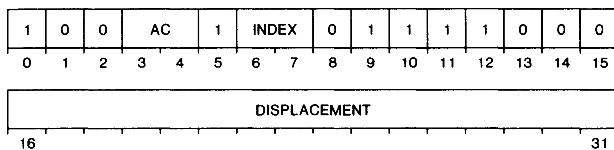
Copies a word from memory to an accumulator.

Calculates the effective address (*E*). Places the contents of the location addressed by *E* in the specified accumulator. The contents of the location addressed by *E* remain unchanged.

ELDB

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 8–15 of the specified accumulator. The instruction sets bits 0–7 of the specified accumulator to 0.

The instruction destroys the previous contents 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. Its values can range from 0 to 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 AC2
1 1	Contents of AC3

Example

ELDB 1,740

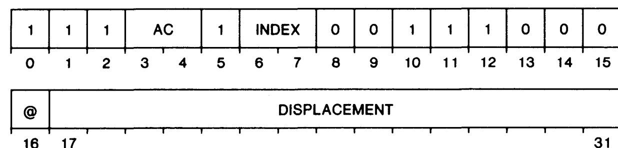
Places the low byte of location 000740_8 into AC1. The parameters of the operation follow.

Parameter	Before	After
Location 0000740_8	016753_8	016753_8
AC1	000000_8	000353_8

ELEF

Extended Load Effective Address

ELEF *ac[@]displacement[,index]*



Places an effective address in an accumulator.

Computes the 15-bit effective address and places it in bits 1–15 of the specified accumulator. Sets bit 0 of the accumulator to 0.

Examples

ELEF 0,TABLE

Places the address of TABLE in AC0. The parameters of the operation follow.

Parameter	Before	After
AC0	Unknown	Logical address of TABLE

ELEF 1,−55,3

Subtracts 000055_8 from the unsigned integer in AC3 and places the result in AC1. The parameters of the operation follow.

Parameter	Before	After
AC1	Unknown	000121_8
AC3	000176_8	000176_8

ELEF 0,.+0

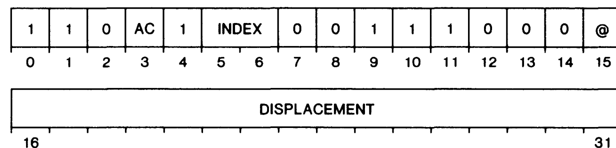
Places the address of this ELEF instruction in AC0. The parameters of the operation follow.

Parameter	Before	After
AC0	Unknown	Logical address of this ELEF instruction

ESTA

Extended Store Accumulator

ESTA *ac[@]displacement[,index]*



Stores the contents of an accumulator in memory.

Calculates the effective address (*E*). Places the contents of the specified accumulator in the word addressed by *E*. The previous contents of the location addressed by *E* are lost. The contents of the specified accumulator remain unchanged.

Example

ESTA 1,1123

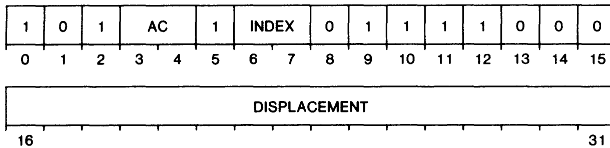
Stores the contents of AC1 in location 001123_8. The parameters of the operation follow.

Parameter	Before	After
AC1	010101_8	010101_8
Location 001123_8	Unknown	010101_8

ESTB

Extended Store Byte

ESTB *ac,displacement[,index]*



Stores in memory the byte contained in 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 8–15 of the specified accumulator.

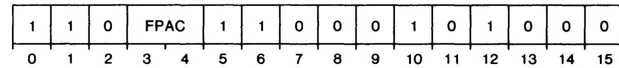
The argument *index* selects the source of the index value. Its values can range from 0 to 3. The following list gives the meaning of each value.

Index Bits	Index Value
0 0	0
0 1	Address of the displacement field (word 2 of this instruction)
1 0	Contents of AC2
1 1	Contents of AC3

FAB

Absolute Value

FAB *fpac*



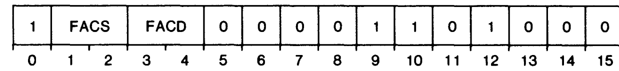
Forms the absolute value of the signed number in a floating-point accumulator (FPAC).

Sets the sign bit of the specified FPAC to 0. Sets the exponent to 0 if the mantissa is 0; otherwise, leaves bits 1–63 of the FPAC unchanged. Updates the Zero (Z) and Negative (N) flags in the floating-point status register to reflect the new contents of the FPAC.

FAD

Add Double (FPAC to FPAC)

FAD *facs,facd*



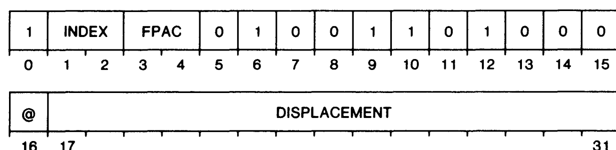
Adds two 64-bit floating-point numbers that reside in floating-point accumulators (FACD, FACS); normalizes the result.

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 Zero (Z) and Negative (N) flags in the floating-point status register to reflect the new contents of FACD.

FAMD

Add Double (Memory to FPAC)

FAMD *fpac[@]displacement[,index]*



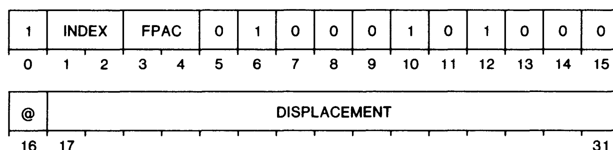
Adds one 64-bit floating-point number in memory to another 64-bit floating-point number in a floating-point accumulator (FPAC) and normalizes the result.

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 Zero (Z) and Negative (N) flags in the floating-point status register to reflect the new contents of the FPAC.

FAMS

Add Single (Memory to FPAC)

FAMS *fpac[@]displacement[,index]*



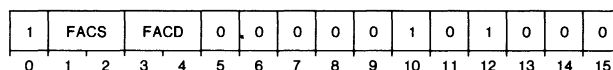
Adds one 32-bit floating-point number in memory to another 32-bit floating-point number in a floating-point accumulator (FPAC) and normalizes the result.

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 Zero (Z) and Negative (N) flags in the floating-point status register to reflect the new contents of the FPAC. Sets bits 32–63 of FPAC to 0.

FAS

Add Single (FPAC to FPAC)

FAS *facs,facd*

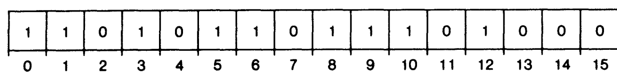


Adds two 32-bit floating-point numbers that reside in floating-point accumulators (FACS, FACD); normalizes the result.

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. Places the normalized result in FACD. Leaves the contents of FACS unchanged. Sets bits 32–63 of FACD to 0 and updates the Zero (Z) and Negative (N) flags in the floating-point status register to reflect the new contents of FACD.

FCLE

Clear Errors



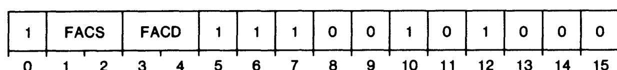
Sets bits 0–4 of the floating-point status register (FPSR) to 0.

NOTES: *The* I/O Reset *(IORST) instruction also sets these bits to 0.*

FCMP

Compare Floating-Point

FCMP *facs,facd*



Compares two 64-bit floating-point numbers that reside in floating-point accumulators (FACS, FACD).

Algebraically compares the floating-point numbers in FACS and FACD to each other. Updates the Zero (Z) and Negative (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 comparison and the corresponding flag settings.

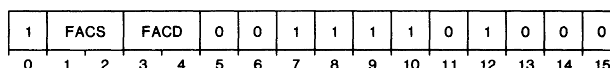
Z	N	Result
1	0	FACS=FACD
0	1	FACS>FACD
0	0	FACS<FACD

NOTE: *Unnormalized operands give unspecified results.*

FDD

Divide Double (FPAC by FPAC)

FDD *facs,facd*



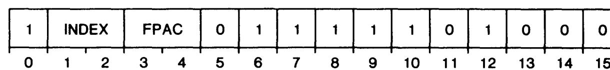
Divides a 64-bit floating-point numbers in a floating-point accumulator (FPAC) by a 64-bit floating-point number in another FPAC; normalizes the result.

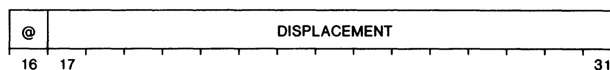
Divides the 64-bit floating-point number in FACD by the 64-bit floating-point number in FACS. Places the normalized results in FACD. Destroys the previous contents of FACD. Leaves the contents of FACS unchanged and updates the Zero (Z) and Negative (N) flags in the floating-point status register to reflect the new contents of FACD.

FDMD

Divide Double (FPAC by Memory)

FDMD *fpac[@]displacement[,index]*





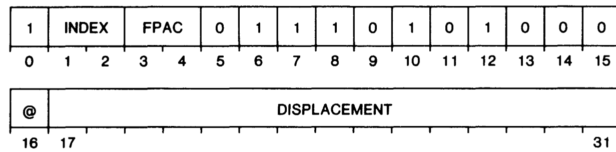
Divides a 64-bit floating-point number in a floating-point accumulator (FPAC) by a 64-bit floating-point number in memory and normalizes the result.

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 Zero (Z) and Negative (N) flags in the floating-point status register to reflect the new contents of the FPAC.

FDMS

Divide Single (FPAC by Memory)

FDMS *fpac[@]displacement[,index]*



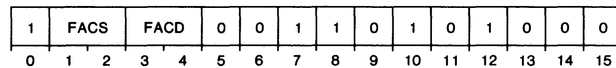
Divides a 32-bit floating-point number in a floating-point accumulator (FPAC) by a 32-bit floating-point number in memory and normalizes the result.

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 Zero (Z) and Negative (N) flags in the floating-point status register to reflect the new contents of the FPAC. Sets bits 32–63 of FACD to 0.

FDS

Divide Single (FPAC by FPAC)

FDS *facs,facd*



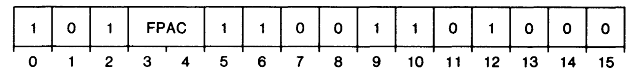
Divides a 32-bit floating-point number in a floating-point accumulator (FPAC) by a 32-bit floating-point number in another FPAC; normalizes the result.

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 Zero (Z) and Negative (N) flags in the floating-point status register to reflect the new contents of FACD. Sets bits 32–63 of FACD to 0.

FEXP

Load Exponent

FEXP *fpac*



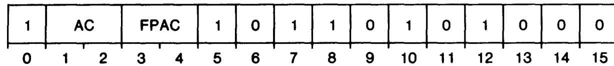
Loads an exponent into a floating-point accumulator (FPAC).

Places bits 1–7 of AC0 in bits 1–7 of the specified FPAC. Ignores bits 0 and 8–15 of AC0. Leaves bits 0 and 8-63 of FPAC and the contents of AC0 unchanged. Sets bits 0-7 (sign and exponent) to 0 if bits 8-63 (mantissa) of FPAC are 0. If FPAC contains true zero, the instruction does not load bits 1–7 of the FPAC.

FFAS

Fix to AC

FFAS *ac,fpac*



Converts the integer portion of a 64-bit floating-point number in a floating-point accumulator (FPAC) into a signed two's complement integer.

Forms the absolute value of the integer portion of the floating-point number in the specified FPAC. Extracts the 15 least significant bits from this value and, if the number in FPAC is negative, forms the two's complement of the integer. Places the result in the specified accumulator and sets the Zero (Z) and Negative (N) flags in the floating-point status register to 0. The contents of the FPAC remain unchanged.

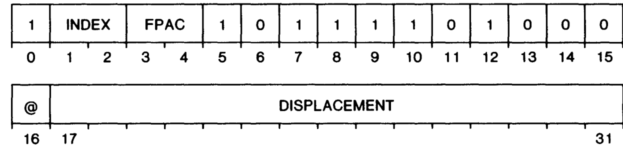
If the number in FPAC is less than −32,768 or greater than +32,767, this instruction sets the Mantissa Overflow (MOF) flag in the floating-point status register to 1.

NOTE: *If the least significant 15 bits of the integer formed from the number in the FPAC are all 0, the sign bit of the result will be 0, regardless of the sign of the original number, unless the number is equal to −32,768.*

FFMD

Fix to Memory

FFMD *fpac[@]displacement[,index]*



Converts the integer portion of a floating-point number in a floating-point accumulator (FPAC) into a signed two's complement integer; stores the result in memory.

Forms the absolute value of the integer portion of the floating-point number in the specified FPAC. Extracts the 31 least significant bits from this value and, if the number in FPAC is negative, forms the two's complement of the integer. Then calculates the effective address (E) and stores the result in the memory locations addressed by E. Sets the Zero (Z) and Negative (N) flags in the floating-point status register to 0. The contents of the FPAC remain unchanged.

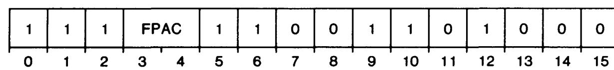
If the number in FPAC is less than −2,147,483,648 or greater than +2,147,483,647 this instruction sets the Mantissa Overflow (MOF) flag in the floating-point status register to 1.

NOTE: *If the least significant 31 bits of the integer formed from the number in the FPAC are all 0, the sign bit of the result will be 0, regardless of the sign of the original number, unless the number is equal to −2,147,483,648.*

FHLV

Halve

FHLV *fpac*



Divides a 64-bit floating-point number in a floating-point accumulator (FPAC) by 2 and normalizes the result.

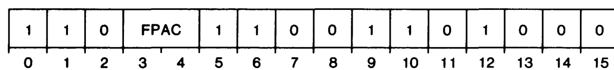
Shifts the mantissa contained in the specified FPAC to the right one bit position. Fills the vacated bit position with a zero and places the bit shifted out in the guard digit. Normalizes the number and places the result in the FPAC. Updates the Zero (Z) and Negative (N) flags in the floating-point status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

If underflow occurs, sets the Exponent Underflow (UNF) flag in the floating-point status register to 1. In this case, the mantissa and sign in the FPAC are correct, but the exponent is 128 too large.

FINT

Integerize

FINT *fpac*



Converts a 64-bit floating-point number in a floating-point accumulator (FAPC) into an integer and normalizes the result.

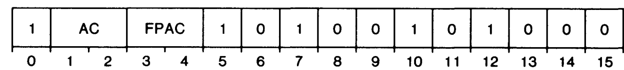
Zeros the fractional portion (if any) of the number contained in the specified FPAC. Normalizes the result. Updates the Zero (Z) and Negative (N) flags in the floating-point status register to reflect the new contents of the specified FPAC.

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.*

FLAS

Float From AC

FLAS *ac,fpac*



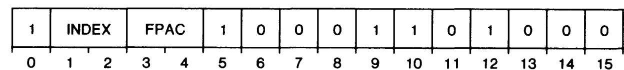
Converts a two's complement ranging from −32,768 to +32,767, inclusive, to floating-point format.

Converts the signed two's complement contained in the specified accumulator to a single-precision floating-point number. Places the result in the most significant 32 bits of the specified FPAC. Sets the least significant 32 bits of the FPAC to 0. Updates the Zero (Z) and Negative (N) flags in the floating-point status register to reflect the new contents of the FPAC. The contents of the specified accumulator remain unchanged.

FLDD

Load Floating-Point Double

FLDD *fpac[@]displacement[,index]*



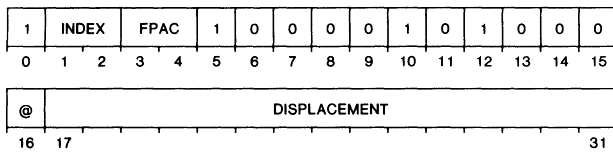
Moves four words out of memory and into a floating-point accumulator (FPAC).

Computes the effective address (*E*). Fetches the double-precision floating-point number at the address specified by *E* and places it in the specified FPAC. Sets the sign and exponent to 0 if the mantissa is 0. Updates the Zero (Z) and Negative (N) flags in the floating-point status register to reflect the new contents of the FPAC.

FLDS

Load Floating-Point Single

FLDS *fpac[@]displacement[,index]*



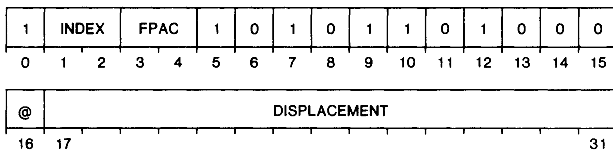
Moves two words out of memory into a floating-point accumulator (FPAC).

Computes the effective address (*E*). Fetches the single-precision floating-point number at the address specified by *E*. Places the number in the FPAC. Sets the sign and exponent to 0 if the mantissa is 0. Updates the Zero (Z) and Negative (N) flags in the floating-point status register to reflect the new contents of the FPAC.

FLMD

Float From Memory

FLMD *fpac[@]displacement[,index]*



Converts the contents of memory locations into floating-point format.

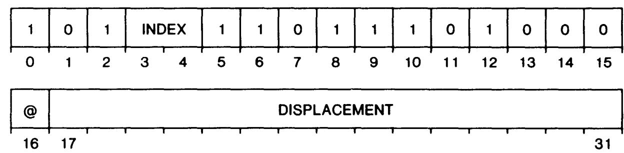
Computes the effective address (*E*). Converts the 32-bit signed two's complement addressed by *E* to a double-precision floating-point number. Places the result in the specified FPAC. Updates the Zero (Z) and Negative (N) flags in the floating-point status register to reflect the new contents of the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

The range of numbers you can convert is −2,147,483,648 to +2,147,483,647, inclusive.

FLST

Load Floating-Point Status

FLST *[@]displacement[,index]*



Moves two words out of memory into the floating-point status register (FPSR).

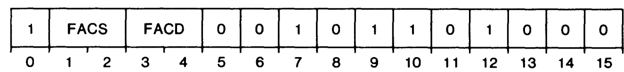
Computes the effective address (*E*). Places the 32-bit operand addressed by *E* in the FPSR and sets the condition codes to the values of the loaded bits.

> **NOTE:** *Bits 12–15 of the FPSR are not set from memory. These bits are the floating-point identification code and cannot be changed. For the specific code, refer to the assembly language programming manual for the particular computer.*

FMD

Multiply Double (FPAC by FPAC)

FMD *facs,facd*



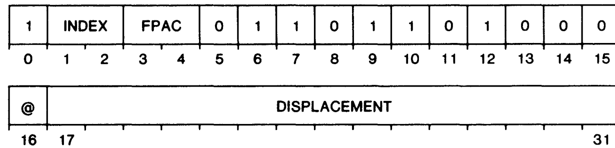
Multiplies two 64-bit floating-point numbers, both of which reside in floating-point accumulators, and normalizes the result.

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 Zero (Z) and Negative (N) flags in the floating-point status register to reflect the new contents of FACD.

FMMD

Multiply Double (FPAC by Memory)

FMMD *fpac[@]displacement[,index]*



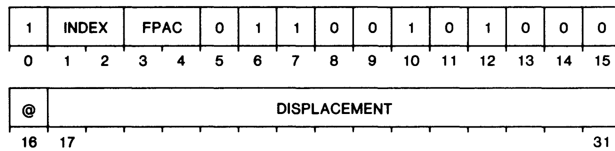
Multiplies one 64-bit floating-point number in a floating-point accumulator (FPAC) by another 64-bit floating-point number in memory and normalizes the result.

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 Zero (Z) and Negative(N) flags in the floating-point status register to reflect the new contents of the FPAC.

FMMS

Multiply Single (FPAC by Memory)

FMMS *fpac[@]displacement[,index]*



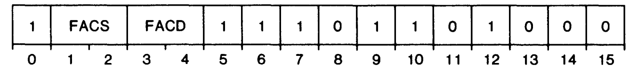
Multiplies a 32-bit floating-point number in a floating-point accumulator by a 32-bit floating-point number in memory and normalizes the result.

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 the specified FPAC. Places the normalized result in the specified FPAC. Leaves the contents of the source location unchanged and updates the Zero (Z) and Negative (N) flags in the floating-point status register to reflect the new contents of the FPAC.

FMOV

Move Floating-Point

FMOV *facs,facd*



Moves the 64-bit contents of one floating-point accumulator into another.

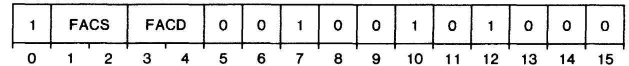
Places the contents of FACS in FACD. Sets the sign and exponent to 0 in FACD if the mantissa in FACS is 0. Updates the Zero (Z) and Negative (N) flags in the floating-point status register to reflect the new contents of FACD. The contents of FACS remain unchanged.

> **NOTE:** *This instruction will move unnormalized data without change, but the Z and N flags will be undefined.*

FMS

Multiply Single (FPAC by FPAC)

FMS *facs,facd*



Multiplies two 32-bit floating-point numbers that reside in floating-point accumulators; normalizes the result.

Multiplies the 32-bit floating-point number in FACS by the 32-bit floating-point number in FACD. Places the normalized result in FACD. Leaves the contents of FACS unchanged. Updates the Zero (Z) and Negative (N) flags in the floating-point status register to reflect the new contents of FACD.

FNEG
Negate
FNEG *fpac*

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

Changes the sign bit of a 64-bit number in a floating-point accumulator (FPAC).

Inverts bit 0 (sign bit) of the specified FPAC and leaves bits 1–63 of the FPAC unchanged. Sets the sign and exponent to 0 if the mantissa is 0. Updates the Zero (Z) and Negative (N) flags in the floating-point status register to reflect the new contents of the FPAC.

If the FPAC contains true zero, leaves the sign (bit 0) unchanged.

FNOM
Normalize
FNOM *fpac*

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

Normalizes the floating-point number in a floating-point accumulator (FPAC).

Sets a true zero in the specified FPAC if all the bits of the mantissa are zero. Updates the Zero (Z) and Negative (N) flags in the floating-point status register to reflect the new contents of the FPAC.

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

FNS
No Skip

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

Executes the next sequential word.

FPOP
Pop Floating-Point State

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

Pops the state of the floating-point processor off the stack.

Pops an 18-word return block off the stack and loads the contents into the floating-point status register (FPSR) and the four floating-point accumulators (FPACs). The format of the 18-word block is shown in Figure 8.3.

NOTES: *Because of the potentially long time required to perform some floating-point instructions in relation to I/O interrupt requests, these instructions are interruptible. Because the FACD, stack pointer, and program counter are not updated until the completion of these instructions, any interrupt service routines that return control to the interrupted program via the program counter stored in location 0 will correctly restart these instructions*

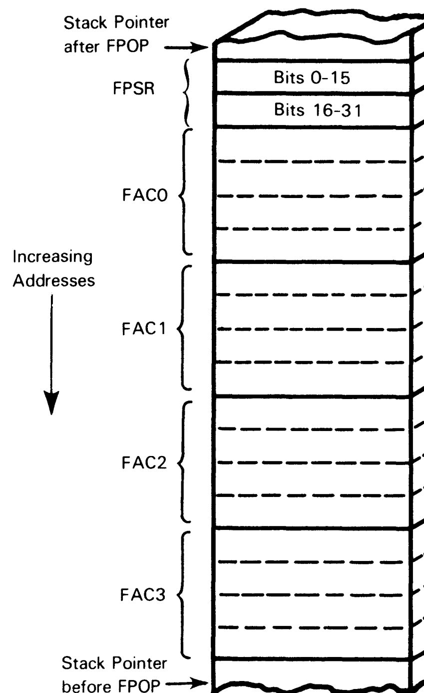


Figure 8.3 Format of 18-word return block

FPSH
Push Floating-Point State

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

Pushes the state of the floating-point processor on the stack.

Pushes an 18-word floating-point return block onto the stack, leaving the contents of the floating-point accumulators and the floating-point status register (FPSR) unchanged. The format of the 18 words pushed is illustrated in Figure 8.4.

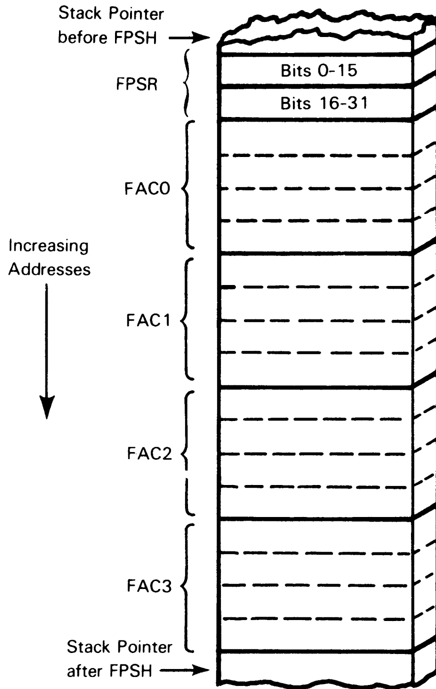


Figure 8.4 Format of 18-word return block pushed on stack

FRH
Read High Word
FRH *fpac*

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

Moves the most significant 16 bits from a floating-point accumulator (FPAC) to an accumulator.

Places the most significant 16 bits of the specified FPAC in AC0. The FPAC and the floating-point status register remain unchanged.

FSA
Skip Always

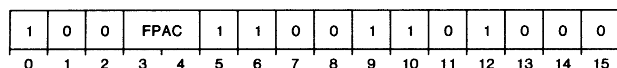
1	0	0	0	1	1	1	0	1	0	1	0	1	0	0	0
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Skips the next sequential word.

FSCAL

Scale

FSCAL *fpac*



Shifts the mantissa of the 64-bit floating-point number in a floating-pont accumulator (FPAC) and replaces its exponent.

Subtracts the exponent in bits 1–7 of the specified FPAC from the exponent in bits 1–7 of AC0. The difference between the exponents specifies *D*, a number of hex digits.

If *D* is 0, the instruction stops.

If *D* is positive, the instruction shifts the mantissa of the number contained in the FPAC to the right by *D* digits.

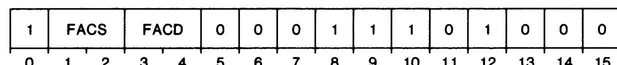
If *D* is negative, the instruction shifts the mantissa of the number contained in the FPAC to the left by *D* digits. Sets the Mantissa Overflow (MOF) flag to 1 in the floating-point status register.

After the shift, the instruction loads the contents of bits 1–7 of AC0 into the exponent field of the FPAC. Bits shifted out of either end of the mantissa are lost. If the entire mantissa is shifted out of the FPAC, the instruction sets the FPAC to true zero. The instruction updates the Zero (Z) and Negative (N) flags in the floating-point status register to reflect the new contents of the FPAC. AC0 remains unchanged.

FSD

Subtract Double (FPAC From FPAC)

FSD *facs,facd*

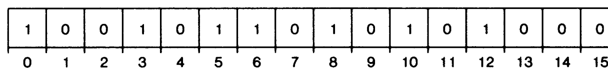


Subtracts a 64-bit floating-point number in a floating-point accumulator (FPAC) from a 64-bit floating-point number in another FPAC; normalizes the result.

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 Zero (Z) and Negative (N) flags in the floating-point status register to reflect the new contents of FACD. The contents of FACS remain unchanged.

FSEQ

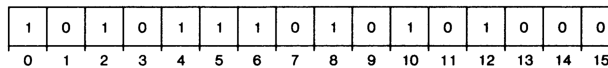
Skip on Zero



Skips the next sequential word if the Zero (Z) flag of the floating-point status register is 1.

FSGE

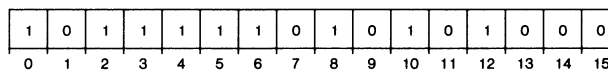
Skip on Greater Than or Equal To Zero



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

FSGT

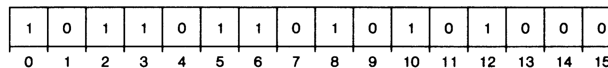
Skip on Greater Than Zero



Skips the next sequential word if both the Zero (Z) and Negative (N) flags of the floating-point status register are 0.

FSLE

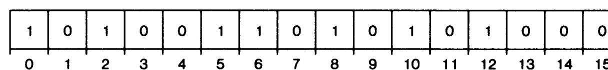
Skip on Less Than or Equal To Zero



Skips the next sequential word if either the Zero (Z) or Negative (N) flag of the floating-point status register is 1.

FSLT

Skip on Less Than Zero

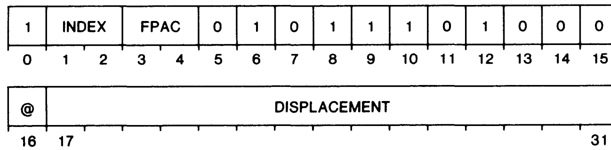


Skips the next sequential word if the Negative (N) flag of the floating-point status register is 1.

FSMD

Subtract Double (Memory From FPAC)

FSMD *fpac[@]displacement[,index]*



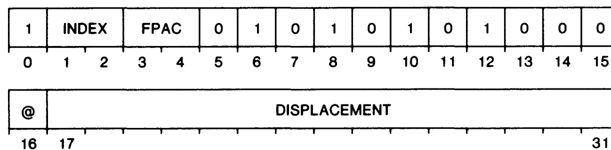
Subtracts a 64-bit floating-point number in memory from a 64-bit floating-point number in a floating-point accumulator (FPAC) and normalizes the result.

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 Zero (Z) and Negative (N) flags in the floating-point status register to reflect the new contents of the FPAC.

FSMS

Subtract Single (Memory From FPAC)

FSMS *fpac[@]displacement[,index]*

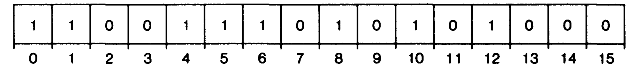


Subtracts a 32-bit floating-point number in memory from a 32-bit floating-point number in a floating-point accumulator (FPAC) and normalizes the result.

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 the specified FPAC. Places the normalized result in the specified FPAC. Leaves the contents of the source location unchanged and updates the Zero (Z) and Negative (N) flags in the floating-point status register to reflect the new contents of the FPAC.

FSND

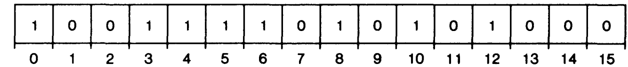
Skip on No Zero Divide



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

FSNE

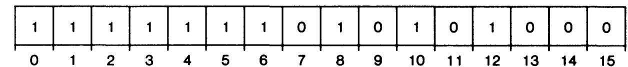
Skip on Nonzero



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

FSNER

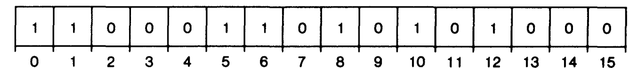
Skip on No Error



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

FSNM

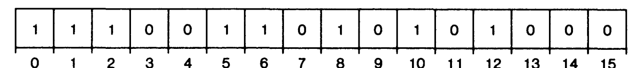
Skip on No Mantissa Overflow



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

FSNO

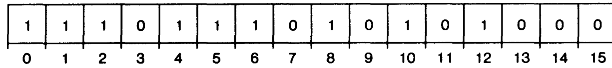
Skip on No Overflow



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

FSNOD

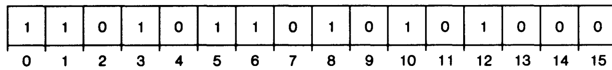
Skip on No Overflow and No Zero Divide



Skips the next sequential word if both the Exponent Overflow (OVF) flag and the Divide By Zero (DVZ) flag of the floating-point status register are 0.

FSNU

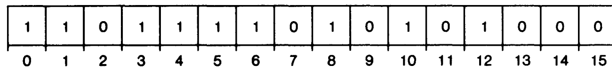
Skip on No Underflow



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

FSNUD

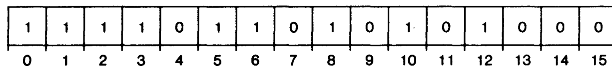
Skip on No Underflow and No Zero Divide



Skips the next sequential word if both the Exponent Underflow (UNF) flag and the Divide by Zero (DVZ) flag of the floating-point status register are 0.

FSNUO

Skip on No Underflow and No Overflow

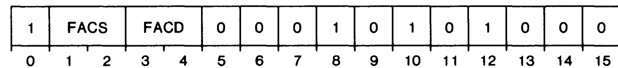


Skips the next sequential word if both the Exponent Underflow (UNF) flag and the Exponent Overflow (OVF) flag of the floating-point status register are 0.

FSS

Subtract Single (FPAC from FPAC)

FSS *facs,facd*



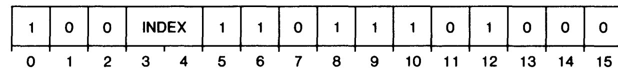
Subtracts a 32-bit floating-point number in a floating-point accumulator (FPAC) from a 32-bit floating-point number in another FPAC; normalizes the results.

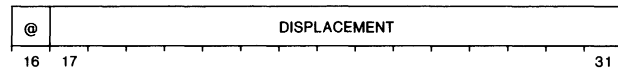
Subtracts the 32-bit floating-point number in FACS from the 32-bit floating-point number in FACD. Places the normalized result in FACD. Updates the Zero (Z) and Negative (N) flags in the floating-point status register to reflect the new contents of FACD. The contents of FACS remain unchanged.

FSST

Store Floating-Point Status

FSST *[@]displacement[,index]*



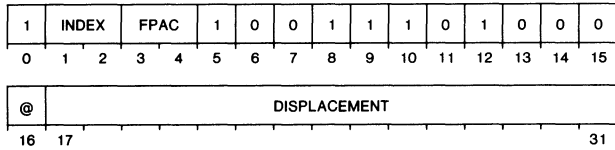


Moves the contents of the floating-point status register (FPSR) to memory.

Computes the effective address (*E*) of two sequential 16-bit locations in memory. Stores the contents of the FPSR in these locations.

FSTD
Store Floating-Point Double

FSTD *fpac[@]displacement[,index]*

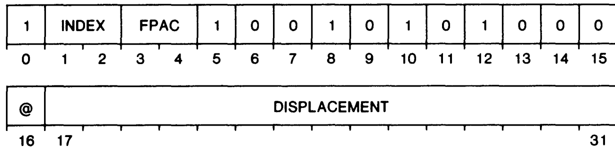


Stores the 64-bit contents of a floating-point accumulator (FPAC) in memory.

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

FSTS
Store Floating-Point Single

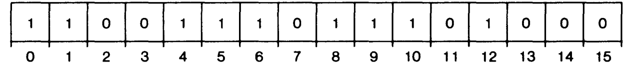
FSTS *fpac[@]displacement[,index]*



Stores 32 bits of a floating-point accumulator (FPAC) in memory.

Computes the effective address (*E*). Places the 32 most significant bits of the specified FPAC in memory beginning at the location addressed by *E*. Destroys the previous contents of the addressed memory location. The contents of the FPAC and the condition codes in the FPSR remain unchanged.

FTD
Trap Disable

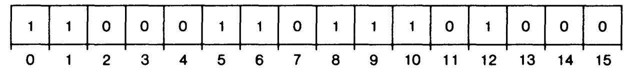


Disables traps on floating-point faults.

Sets the Trap Enable (TE) bit of the floating-point status register to 0.

> **NOTE:** *The* I/O Reset *(IORST) instruction also sets this bit to 0.*

FTE
Trap Enable



Enables traps on floating-point faults.

Sets the Trap Enable (TE) bit of the floating-point status register (FPSR) to 1.

> **NOTES:** *When this instruction is used to cause a floating-point trap, the floating-point program counter field (FPPC) of the FPSR contains the address of the instruction to cause a fault.*
>
> *When a floating-point fault occurs and the TE flag is 1, the processor sets the TE flag to 0 before transferring control to the floating-point fault handler. The fault handler should set the TE flag to 1 before normal processing is resumed.*

HALT

Halt

DOC*[f]* ***ac*,CPU**

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

Stops program execution.

The DOC mnemonic sets the Interrupt On (ION) flag according to the function specified by *f* and then stops the processor. You must specify an accumulator with this mnemonic, even though its contents are ignored.

DGC assemblers recognize the HALT mnemonic as equivalent to DOC 0, CPU.

HLV

Halve

HLV ***ac***

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

Divides the contents of an accumulator by 2 and rounds the result toward 0.

Divides the signed, 16-bit two's complement in the specified accumulator by 2. Rounds the result toward 0 and stores it in the specified accumulator.

Examples

HLV 0

Divides the contents of AC0 by 2. The parameters of the operation follow.

Parameter	Before	After
AC0	004523_8	002251_8

HLV 1

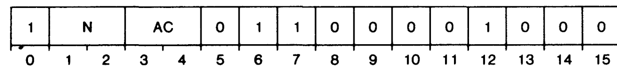
Divides the contents of AC1 by 2. The parameters of the operation follow.

Parameter	Before	After
AC1	177776_8	177777_8

HXL

Hex Shift Left

HXL *n,ac*



Logically shifts the contents of an accumulator left 1 to 4 hex digits.

Shifts the contents of the specified accumulator a number of hex digits to the left. The number of digits shifted depends on the immediate field N and equals N+1. Bits shifted out are lost, and the bit positions vacated are filled with zeros. If N equals 3, then all the bits in the specified accumulator are shifted out and set to 0.

> **NOTE:** *DGC assemblers subtract one from the coded value of* n *before placing it in the immediate field. Therefore, you should code the exact number of hex digits you want shifted.*

Example

HXL 2,1

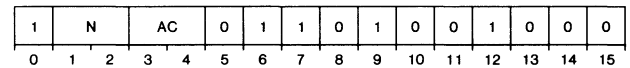
Shifts the number in AC1 left 2 hex digits. The parameters of the operation follow.

Parameter	Before	After
AC1	004523_8	051400_8

HXR

Hex Shift Right

HXR *n,ac*



Logically shifts the contents of an accumulator 1 to 4 hex digits right.

Shifts the contents of the specified accumulator a number of hex digits to the right. The number of digits shifted depends on the immediate field N and equals N+1. Bits shifted out are lost, and the bit positions vacated are filled with zeros. If N equals 3, then all bits in the specified accumulator are shifted out and set to 0.

> **NOTE:** *DGC assemblers subtract one from the coded value of* n *before placing it in the immediate field. Therefore, you should code the exact number of hex digits you want shifted.*

Example

HXR 2,1

Shifts the number in AC1 right 2 hex digits. The parameters of the operation follow.

Parameter	Before	After
AC1	004523_8	000011_8

INC

Increment

INC*[c][sh][#]* *acs,acd[,skip]*

1	ACS	ACD	0	1	1	SH	C	#	SKIP
0	1–2	3–4	5	6	7	8–9	10–11	12	13–15

Increments the contents of an accumulator.

Initializes carry to the specified value. Increments the unsigned, 16-bit number in ACS by one and places the result in the shifter. If the result of the incrementation exceeds 32,768, the instruction complements the carry bit. Performs the specified shift operation and loads the result of the shift into ACD if the no-load bit is 0. If the skip condition is true, the next sequential word is skipped.

Options

[c]

The processor determines the effect of carry (*c*) on the old value of Carry before performing the operation (*opcode*). The following table lists the values of *c* and bits 10 and 11 and describes the operation.

Symbol *[c]*	Bits 10-11	Operation
Omitted	0 0	Leaves carry unchanged
Z	0 1	Initializes carry to 0
O	1 0	Initializes carry to 1
C	1 1	Complements carry

[sh]

The processor shifts carry 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 table lists the values of *sh* and bits 8 and 9 and describes the shift operation.

Symbol *[sh]*	Bits 8-9	Shift Operation
Omitted	0 0	Does not shift the result
L	0 1	Shifts left
R	1 0	Shifts right
S	1 1	Swaps 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 table lists the values of the no-load option and bit 12 and describes the operation.

Symbol *[#]*	Bit 12	Operation
Omitted	0	Loads the result into ACD
#	1	Does not load the result; restores the carry bit

NOTE: *You can not combine the no-load option (#) with either the never skip or always skip option because the bit patterns for these combinations are used to define other types of instructions. This means that the ADC instruction cannot end in* 1000_2 *or* 1001_2.

[skip]

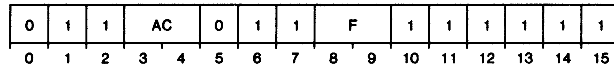
The processor can skip the next instruction if the condition test is true. The following table lists the test conditions and the values of bits 13–15 and describes the operation.

Symbol *[skip]*	Bits 13-15	Operation
Omitted	0 0 0	Never skips
SKP	0 0 1	Always skips
SZC	0 1 0	Skips if carry is 0
SNC	0 1 1	Skips if carry is not 0
SZR	1 0 0	Skips if the result is 0
SNR	1 0 1	Skips if the result is not 0
SEZ	1 1 0	Skips if either carry or the result is 0
SBN	1 1 1	Skips if both carry and the result are not 0

When the instruction skips, it skips the next sequential 16-bit word. Be certain that a skip does not transfer control to the middle of an instruction that is 32-bits long.

INTA
Interrupt Acknowledge

INTA *ac*
DIB*[f]* *ac*,**CPU**



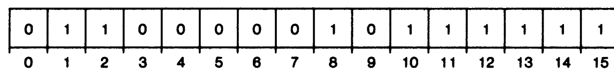
Places a 6-bit device code in bits 10–15 of the specified accumulator. This device code identifies the highest priority device currently requesting an interrupt. The value of bits 0–9 of the specified accumulator depend on the specific computer.

After the transfer, the DIB mnemonic sets the Interrupt On (ION) flag according to the function specified by *f*.

The INTA mnemonic does not affect the ION flag when it places the device code into bits 10–15 of the specified accumulator.

INTDS
Interrupt Disable

INTDS
NIOC CPU

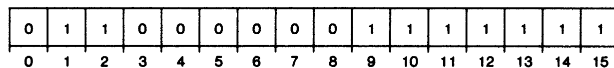


Disables interrupt facility.

Sets the Interrupt On (ION) flag to 0.

INTEN
Interrupt Enable

INTEN
NIOS CPU



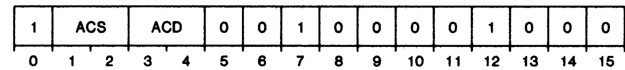
Enables the interrupt facility.

Sets the Interrupt On (ION) flag to one.

If the instruction changes the state of the ION flag, the processor allows one more instruction to execute before the first I/O interrupt can occur. However, if the instruction is interruptible, then interrupts can occur as soon as the instruction begins to execute.

IOR
Inclusive OR

IOR *acs,acd*



Inclusively ORs the contents of two accumulators.

Forms the logical inclusive OR of the contents of ACS and the contents of ACD, and places the result in 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 are unchanged.

Example

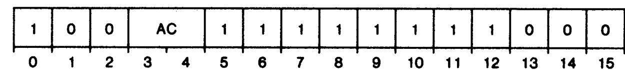
```
IOR     0,1
```

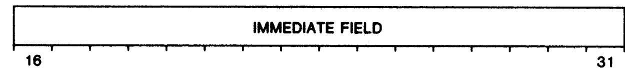
Inclusively ORs the contents of AC0 with the contents of AC1. The parameters of the operation follow.

Parameter	Before	After
AC0	002245_8	002245_8
AC1	010133_8	012377_8

IORI
Inclusive OR Immediate

IORI *i,ac*





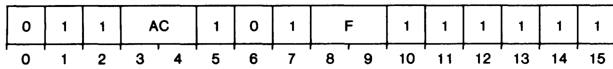
Inclusively ORs the contents of an accumulator with a number in the instruction.

Forms the logical inclusive OR of the contents of the immediate field and the contents of the specified accumulator, and places the result in the specified accumulator.

IORST

I/O Reset

IORST
DIC*[f]* *ac*,**CPU**



Resets the I/O system.

Sends a reset signal to all devices to clear their states. Sets device Busy and Done flags to 0.

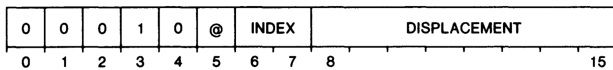
The IORST mnemonic sets the Interrupt On (ION) flag and the 16-bit priority mask to 0.

DGC assemblers recognize the mnemonic IORST as equivalent to DICC 0,CPU. The DIC mnemonic sets the ION flag according to the function specified by *f* and sets the 16-bit priority mask to 0. When using this mnemonic, you must code an accumulator to avoid an assembly error, even though the accumulator is ignored.

ISZ

Increment and Skip if Zero

ISZ *[@]displacement[,index]*



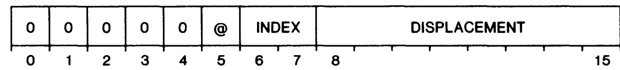
Increments the addressed word, then skips if the incremented value is 0.

Computes the effective address (*E*). Increments the word addressed by *E* by 1 and writes the result back into memory at the addressed location. If the location's updated value is 0, the instruction skips the next sequential word.

JMP

Jump

JMP *[@]displacement[,index]*



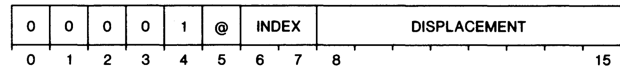
Loads an effective address into the program counter.

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.

JSR

Jump to Subroutine

JSR *[@]displacement[,index]*

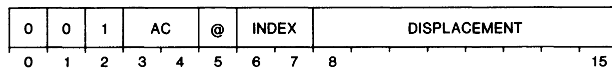


Increments and stores the value of the program counter in an accumulator and then places a new address in the program counter.

Computes the effective address (*E*), places the address of the next sequential instruction in AC3, and places *E* in the program counter. Sequential operation continues with the word addressed by the updated value of the program counter.

LDA
Load Accumulator

LDA *ac[@]displacement[,index]*

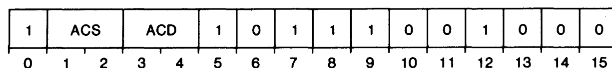


Copies a word from memory to an accumulator.

Calculates the effective address (*E*) and places the word addressed by *E* in the specified accumulator. The contents of the location addressed by *E* are unchanged.

LDB
Load Byte

LDB *acs,acd*

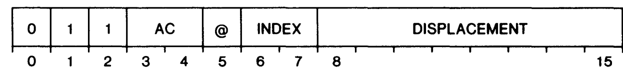


Stores in memory a byte contained in an accumulator.

Places the 8-bit byte addressed by the byte pointer contained in ACS into bits 8–15 of ACD. Sets bits 0–7 of ACD to 0. The contents of ACS remain unchanged unless ACS and ACD are the same accumulator.

LEF
Load Effective Address

LEF *ac[@]displacement[,index]*



Places an effective address in an accumulator.

Computes the effective address (*E*), places it in the specified accumulator, and sets bit 0 of the accumulator to 0. The previous contents of the AC are lost.

The LEF instruction can be executed only when the user address translator is enabled and the LEF flag in the address translator's status register is set to 1. In any other situation, the processor checks the I/O Validity flag in the address translator's status register. If this flag is set to 1 or user address translation is disabled, the processor executes the LEF instruction as an I/O instruction. Otherwise, a protection violation occurs.

Examples

```
LEF    0,TABLE
```

Places the address of TABLE in AC0. The parameters of the operation follow.

Parameter	Before	After
AC0	Unknown	Logical address of TABLE

```
LEF    1,-55,3
```

Subtracts 000055_8 from the unsigned integer in AC3 and places the result in AC1. The parameters of the operation follow.

Parameter	Before	After
AC1	Unknown	000121_8
AC3	000176_8	000176_8

```
LEF    0,.+0
```

Places the address of this LEF instruction in AC0. The parameters of the operation follow.

Parameter	Before	After
AC0	Unknown	Logical address of this ELEF instruction

LMP

Load User/DCH Map Table (Load Map)

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

In computers with address translation, loads successive words from memory into the selected user or data channel map table.

For details, refer to the assembly language programming manual for the specific computer.

LOB

Locate Lead Bit

LOB *acs,acd*

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

Counts the number of most significant zeros in an accumulator.

Adds a number equal to the number of most significant zeros in the contents of ACS to the signed, 16-bit, two's complement contained in ACD. The contents of ACS and carry are unchanged.

> **NOTE:** *If ACS and ACD are specified as to the same accumulator, the instruction functions as described above, except that the contents of ACS are changed.*

LRB

Locate and Reset Lead Bit

LRB *acs,acd*

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

Performs a *Locate Lead Bit* instruction and sets the lead bit to 0.

Adds a number equal to the number of most significant zeros in the contents of ACS to the signed, 16-bit, two's complement contained in ACD. Sets the leading 1 in ACS to 0.

> **NOTE:** *If ACS and ACD are specified are the same accumulator, then the instruction sets the leading 1 in that accumulator to 0 and no count is taken.*

LSH

Logical Shift

LSH *acs,acd*

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

Logically shifts the contents of an accumulator left or right.

Shifts the contents of ACD to the left or right, depending on the number contained in bits 8–15 of ACS. The signed, 8-bit two's complement contained in bits 8–15 of ACS determines the direction of the shift and the number of bits to be shifted. If the number in bits 8–15 of ACS is positive, contents shift left; if the number in bits 8–15 of ACS is negative, contents shift right. If the number in bits 8–15 of ACS is 0, contents do not shift. Bits 0–7 of ACS are ignored.

The number of bits shifted is equal to the magnitude of the number in bits 8–15 of ACS. Bits shifted out are lost and the bit positions vacated are filled with zeros. Carry and the contents of ACS are unchanged.

> **NOTE:** *If the magnitude of the number in bits 8–15 of ACS exceeds 15, all bits of ACD are set to 0. Carry and the contents of ACS remain unchanged.*

Examples

```
LSH    0,1
```

Shifts the contents of AC1. The two different LSH 0,1 examples are given below. The first example shifts the contents of AC1 left one bit. The parameters of the operation follow.

Parameter	Before	After
AC0	000001_8	000001_8
AC1	012345_8	024712_8

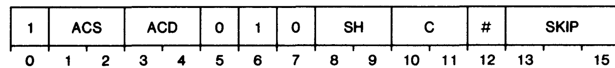
Shifts the contents of AC1 right one bit. The parameters of the operation follow.

Parameter	Before	After
AC0	000377_8	000377_8
AC1	024712_8	012345_8

MOV

Move

MOV*[c][sh][#]*   *acs,acd[,skip]*



Moves the contents of an accumulator into another accumulator.

Initializes carry to the specified value. Places the contents of ACS in the shifter. Performs the specified shift operation and loads the result of the shift into ACD if the no-load bit is 0. If the skip condition is true, the instruction skips the next sequential word.

Options

[c]

The processor determines the effect of carry (*c*) on the old value of carry before performing the operation (*opcode*). The following table lists the values of *c* and bits 10 and 11 and describes the operation.

Symbol *[c]*	Bits 10-11	Operation
Omitted	0 0	Leaves carry unchanged
Z	0 1	Initializes carry to 0
O	1 0	Initializes carry to 1
C	1 1	Complements carry

[sh]

The processor shifts carry 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 table lists the values of *sh* and bits 8 and 9 and describes the shift operation.

Symbol *[sh]*	Bits 8-9	Shift Operation
Omitted	0 0	Does not shift the result
L	0 1	Shifts left
R	1 0	Shifts right
S	1 1	Swaps 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's contents. The following table lists the values of the no-load option and bit 12 and describes the operation.

Symbol *[#]*	Bit 12	Operation
Omitted	0	Loads the result into ACD
#	1	Does not load the result; restores the carry bit

NOTE: *You cannot combine the no-load option (#) with either the never skip or always skip option because the bit patterns for these combinations are used to define other types of instructions. This means that the MOV instruction cannot end in 1000_2 or 1001_2.*

[skip]

The processor can skip the next instruction if the test condition is true. The following table lists the test conditions and the values of bits 13 – 15 and describes the operation.

Symbol *[skip]*	Bits 13-15	Operation
Omitted	0 0 0	Never skips
SKP	0 0 1	Always skips
SZC	0 1 0	Skips if carry is 0
SNC	0 1 1	Skips if carry is not 0
SZR	1 0 0	Skips if the result is 0
SNR	1 0 1	Skips if the result is not 0
SEZ	1 1 0	Skips if either carry or the result is 0
SBN	1 1 1	Skips 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 an instruction that is 32-bits long.

MSKO
Mask Out

MSKO *ac*
DOB[*f*] *ac*,**CPU**

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

Specifies the I/O interrupt priority mask.

Places the contents of the specified accumulator in the priority mask. After the transfer, sets the Interrupt On (ION) flag according to the function specified by *f*. The contents of the specified accumulator remain unchanged.

NOTES: *A 1 in any bit disables interrupt requests for those devices which use that bit as a mask.*

Do not use this instruction when interrupts are enabled.

MSP
Modify Stack Pointer

MSP *ac*

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

Changes the value of the stack pointer and tests for potential overflow.

Adds the signed two's-complement in 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 exceeds the stack limit, the instruction restores the value of location 40 (its original contents before the add) and pushes a standard return block onto the stack. The program counter in the return block contains the address of this MSP 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.

MUL

Unsigned Multiply

1	1	0	0	0	1	1	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

Multiplies the unsigned contents of two accumulators and adds the result to the unsigned contents of a third accumulator.

Multiplies the unsigned, 16-bit number in AC1 by the unsigned, 16-bit number in AC2 to yield an unsigned, 32-bit intermediate result. The unsigned, 16-bit number in AC0 is added to the intermediate result to produce the final result. The final result is an unsigned, 32-bit number which occupies both AC0 and AC1. Bit 0 of AC0 is the most significant bit of the result; bit 15 of AC1 is the least significant bit. The contents of AC2 are unchanged.

Because the result is a double-length number, overflow cannot occur.

Examples

MUL

Four MUL examples are given below. The first multiplies 3 by 2. The parameters of the operation follow.

Parameter	Before	After
AC0	000000_8	000000_8
AC1	000003_8	000006_8
AC2	000002_8	000002_8

Multiplies 077777_8 by 077777_8. The parameter of the operation follow.

Parameter	Before	After
AC0	000000_8	037777_8
AC1	077777_8	000001_8
AC2	077777_8	077777_8

Multiplies 177777_8 by 177777_8. The parameters of the operation follow.

Parameter	Before	After
AC0	000000_8	177776_8
AC1	177777_8	000001_8
AC2	177777_8	077777_8

Multiplies 3 by 2 and adds a remainder of 1. The parameters of the operation follow.

Parameter	Before	After
AC0	000001_8	000000_8
AC1	000003_8	000007_8
AC2	000002_8	000002_8

MULS

Signed Multiply

1	1	0	0	1	1	1	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

Multiplies the signed contents of two accumulators and adds the result to the signed contents of a third accumulator.

Multiplies the signed, 16-bit two's complement in AC1 by the signed, 16-bit two's complement in AC2 to yield a signed, 32-bit two's complement intermediate result. The signed, 16-bit two's complement in AC0 is added to the intermediate result to produce the final result. The final result is a signed, 32-bit two's complement which occupies both AC0 and AC1. Bit 0 of AC0 is the sign bit of the result and bit 15 of AC1 is the least significant bit. The contents of AC2 remain unchanged.

Because the result is a double-length number, overflow cannot occur.

Examples

MULS

Four MULS examples are given below. The first multiplies 3 by 2. The parameters of the operation follow.

Parameter	Before	After
AC0	000000_8	000000_8
AC1	000003_8	000006_8
AC2	000002_8	000002_8

Multiplies 077777_8 by 077777_8. The parameters of the operation follow.

Parameter	Before	After
AC0	000000_8	037777_8
AC1	077777_8	000001_8
AC2	077777_8	077777_8

Multiplies -1 by -1. The parameters of the operation follow.

Parameter	Before	After
AC0	000000_8	000000_8
AC1	177777_8	000001_8
AC2	177777_8	177777_8

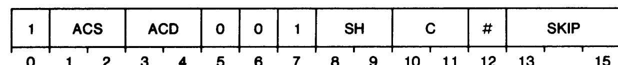
Multiplies 3 by −2 and adds a remainder of −1. The parameters of the operation follow.

Parameter	Before	After
AC0	177777_8	177777_8
AC1	000003_8	177771_8
AC2	177776_8	177776_8

NEG

Negate

NEG*[c][sh][#]*   *acs,acd[,skip]*



Forms the two's complement of an accumulator's contents.

Initializes carry to the specified value. Places the two's complement of the unsigned, 16-bit number in ACS in the shifter. If the negate operation produces a carry of 1 from the most significant bit, the instruction complements carry. Performs the specified shift operation and places the result in ACD if the no-load bit is 0. If the skip condition is true, the instruction skips the next sequential word.

> **NOTE:** *If ACS contains 0, the instruction complements carry.*

Options

[c]

The processor determines the effect of carry (*c*) on the old value of carry before performing the operation (*opcode*). The following table lists the values of c and bits 10 and 11 and describes the operation.

Symbol *[c]*	Bits 10-11	Operation
Omitted	0 0	Leaves carry unchanged
Z	0 1	Initializes carry to 0
O	1 0	Initializes carry to 1
C	1 1	Complements carry

[sh]

The processor shifts carry 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 table lists the values of sh and bits 8 and 9 and describes the shift operation.

Symbol *[sh]*	Bits 8-9	Shift Operation
Omitted	0 0	Does not shift the result
L	0 1	Shifts left
R	1 0	Shifts right
S	1 1	Swaps 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 for testing the result of the instruction operation without destroying the destination accumulator's contents. The following table lists the values of the no-load option and bit 12 and describes the operation.

Symbol *[#]*	Bit 12	Operation
Omitted	0	Loads the result into ACD
#	1	Does not load the result; restores the carry bit

> **NOTE:** *You cannot combine the no-load option (#) with either the never skip or always skip option because the bit patterns for the combinations are used to define other types of instructions. This means that the NEG instruction may not end in 1000_2 or 1001_2.*

[skip]

The processor can skip the next instruction if the condition test is true. The following table lists the test conditions and the values of bits 13 to 15 and describes the operation.

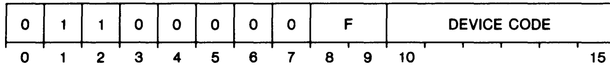
Symbol *[skip]*	Bits 13-15	Operation
Omitted	0 0 0	Never skips
SKP	0 0 1	Always skips
SZC	0 1 0	Skips if carry is 0
SNC	0 1 1	Skips if carry is not 0
SZR	1 0 0	Skips if the result is 0
SNR	1 0 1	Skips if the result is not 0
SEZ	1 1 0	Skips if either carry or the result is 0
SBN	1 1 1	Skips if both carry and the result are not 0

When the instruction skips, it skips the next sequential 16-bit word. Be certain that a skip does not transfer control to the middle of an instruction that is 32-bits long.

NIO

No I/O Transfer

NIO *[f] device*



Changes the state of the Busy and Done flags without performing any other operation.

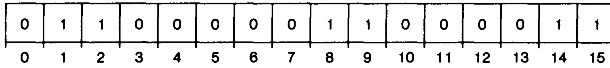
Sets the Busy and Done flags in the specified device according to the function specified by *f*.

NIOP MAP

Translate User Single Cycle
(Map Single Cycle)

or

Disable User Translation
(Disable User Mode)



In a computer with address translation, performs one of the following operations:

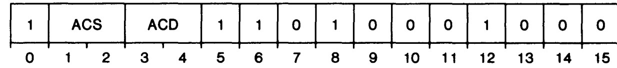
- Translates one user memory reference if issued while user address translation is disabled.
- Disables user address translation if issued while user address translation is enabled and both LEF mode and I/O protection are disabled.

For more information, refer to the assembly language programming manual for the specific computer.

POP

Pop Multiple Accumulators

POP *acs,acd*



Pops one to four words off the stack and places them in accumulators.

The set of accumulators from ACS through ACD is filled with words popped from the stack. The words are popped in descending order, starting with ACS and ending with ACD and wrapping around if necessary, with AC3 following AC0. If ACS equals ACD, only one word is popped, and it is placed in ACS.

The stack pointer is decremented by the number of accumulators popped; the frame pointer is unchanged.

Example

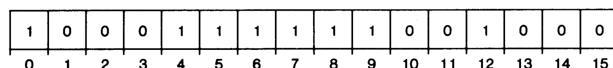
```
POP    3,1
```

Pops three words off the stack and loads them into accumulators AC3 through AC1. The parameters of the operation follow.

Parameter	Before	After
AC0	Unknown	Unchanged
AC1	Unknown	Third word popped
AC2	Unknown	Second word popped
AC3	Unknown	First word popped

POPB

Pop Block



Returns control from an extended operation (XOP or XOP1), system call routine (SYC), or an I/O interrupt handler that does not use the stack change facility of the *Vector* instruction (VCT).

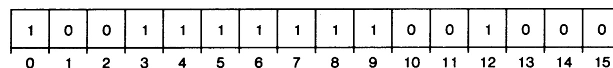
Five words are popped off the stack and placed in predetermined locations. The words popped and their destinations are as follows.

Word Popped	Destination
1	Bit 0 is loaded into carry; bits 1-15 are loaded into the PC
2	AC3
3	AC2
4	AC1
5	AC0

Sequential operation continues with the word addressed by the updated value of the program counter. The frame pointer remains unchanged.

POPJ

Pop PC And Jump



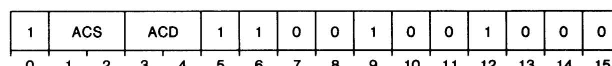
Pops the top word off the stack and places it in the program counter. Continues execution with the word addressed by the updated program counter.

Decrements the stack pointer by 1. The frame pointer remains unchanged.

PSH

Push Multiple Accumulators

PSH *acs,acd*



Pushes the contents of one to four accumulators onto the stack.

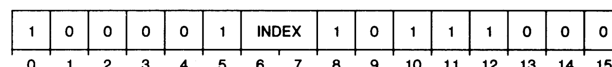
Pushes the set of accumulators from ACS through ACD onto the stack in ascending order. The accumulators are pushed starting with ACS and ending with ACD and wrapping around if necessary, with AC0 following AC3. The contents of the accumulators are unchanged. If ACS equals ACD, only ACS is pushed.

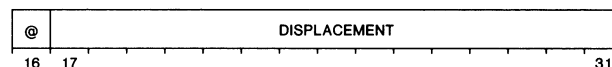
Increments the stack pointer by the number of accumulators pushed. The frame pointer remains unchanged. A check for overflow is made only after the entire push operation is finished.

PSHJ

Push Jump

PSHJ *[@]displacement[,index]*





Pushes the address of the next sequential instruction onto the 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.

PSHR
Push Return Address

1	0	0	0	0	1	1	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

Pushes the address of this instruction *plus 2* onto the stack.

READS
Read Switch Register

READS *ac*
DIA*[f]* *ac*,**CPU**

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

Places the contents of the switch register in an accumulator.

For more information, refer to the assembly language programming manual for the specific computer.

RSTR
Restore

1	1	1	0	1	1	1	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

Returns control from certain types of I/O interrupts.

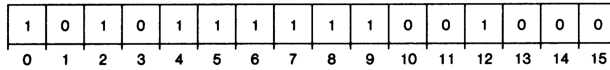
Pops nine words off the stack and places them in predetermined locations. The popped words and their destinations are as follows.

Word Popped	Destination
1	Bit 0 is loaded into carry; bits1-15 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.

> **NOTE:** *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).*

RTN
Return



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* instruction uses this value of the frame pointer to pop a standard return block off of the stack. The format of the return block is as follows.

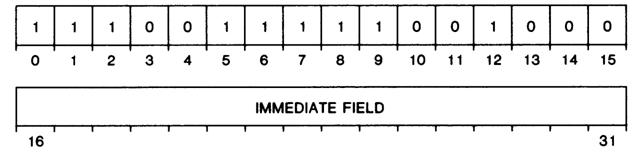
Word Popped	Destination
1	Bit 0 is loaded into carry; bits 1-15 are loaded into the PC
2	AC3
3	AC2
4	AC1
5	AC0

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.

Sequential operation continues with the word addressed by the updated values of the program counter.

SAVE
Save

SAVE *i*



Saves information required by the *Return* instruction.

Using the current stack pointer, pushes a 5-word return block onto the stack and places the current value of the stack pointer (the old stack pointer value plus five) into the frame pointer and AC3. Adds the immediate field to the stack pointer. The contents of AC0, AC1, AC2, and the carry bit are unchanged

The following table shows the format of the 5-word return block.

Word Popped	Destination
1	AC0
2	AC1
3	AC2
4	Frame pointer before the save
5	Bit 0 = carry bit; bits 1-15 = bit 1-15 of AC3

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 is not normally accessed by push and pop operations but is used by the procedure for temporary storage of variables, counters, etc. The frame pointer acts as the reference point for this storage area.

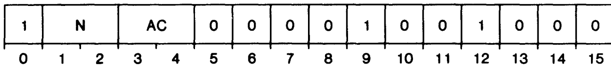
If a stack overflow condition exists, the *Save* instruction transfers control to the stack fault handler. For information on exactly when overflow is checked, refer to the assembly language programming manual for the specific computer.

Use the *Save* instruction with the *Jump to Subroutine* (JSR) instruction which places the return value of the program counter in AC3. *Save* then pushes the return value (contents of AC3) into the fifth word pushed.

SBI

Subtract Immediate

SBI *n,ac*



Subtracts an unsigned integer in the range 1 to 4 from the contents of an accumulator.

Subtracts the value $N+1$ from the unsigned 16-bit number in the specified accumulator and places the result in the accumulator.

> **NOTE:** *DGC assemblers subtract 1 from the coded value of* **n** *before placing it in the immediate field. Therefore, you should code the exact value to be subtracted.*

Example

SBI 4,2

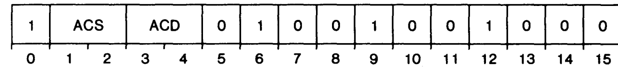
Subtracts 4 from the contents of AC2. The parameters of the operation follow.

Parameter	Before	After
AC2	000003_8	177777_8
Carry	0 or 1	Unchanged

SGE

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.

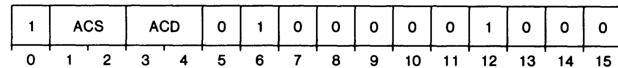
Algebraically compares the signed two's complements in ACS and ACD. If the number in ACS is greater than or equal to the number in ACD, the next sequential word is skipped. The contents of ACS and ASD remain unchanged.

> **NOTE:** *The* Skip if ACS Greater than ACD *(SGT) and* Skip if ACS Greater than or Equal to ACD *(SGE) instructions treat the contents of the specified accumulators as signed, two's complement integers. To compare unsigned integers, use the* Subtract *(SUB) or* Add Complement *(ADC) instruction.*

SGT

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.

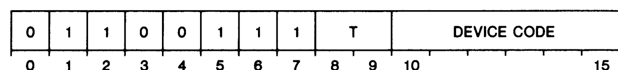
Algebraically compares the signed, two's complements in ACS and ACD. If the number in ACS is greater than the number in ACD, the next sequential word is skipped. The contents of ACS and ACD remain unchanged.

> **NOTE:** *The* Skip if ACS Greater than ACD *(SGT) and* Skip if ACS Greater than or Equal to ACD *(SGE) instructions treat the contents of the specified accumulators as signed, two's complement integers. To compare unsigned integers, use the* Subtract *(SUB) or* Add Complement *(ACD) instruction.*

SKP

I/O Skip

SKP*t* *device*



Tests the Busy and Done flags of a device.

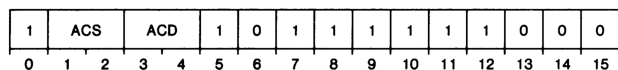
Skips the next sequential word if the test condition specified by *t* is true for the device specified by the device code. The following table lists the test *t* conditions and values and describes the operation.

Symbol *t*	Value	Test
BN	0 0	Skips on Busy = 1
BZ	0 1	Skips on Busy = 0
DN	1 0	Skips on Done = 1
DZ	1 1	Skips on Done = 0

SNB

Skip on Nonzero Bit

SNB *acs,acd*



Skips the next sequential word if the addressed bit is set to 1.

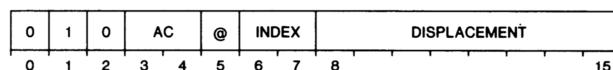
Forms a bit pointer from the contents of ACS and ACD. ACS contains the most significant 16 bits and ACD contains the least significant 16 bits of the bit pointer. If ACS and ACD are specified as the same accumulator, the instruction treats the accumulator contents as the least significant 16 bits of the bit pointer and assumes that the most significant 16 bits are 0.

If the addressed bit in memory is 1, the next sequential word is skipped. The contents of ACS and ACD remain unchanged.

STA

Store Accumulator

STA *ac[@]displacement[,index]*



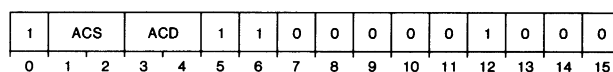
Copies the contents of an accumulator into memory.

Calculates the effective address (*E*) and places the contents of the specified accumulator in the word addressed by *E*. The previous contents of the location addressed by *E* are lost. The contents of the specified accumulator remain unchanged.

STB

Store Byte

STB *acs,acd*



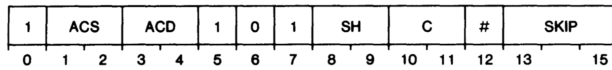
Copies a byte from an accumulator into memory.

Places bits 8–15 of ACD in the byte addressed by the byte pointer contained in ACS. The contents of ACS and ACD remain unchanged.

SUB

Subtract

SUB*[c][sh][#]*   *acs,acd[,skip]*



Performs unsigned integer subtraction and complements carry if appropriate.

Initializes carry to its specified value. Subtracts the unsigned, 16-bit number in ACS from the unsigned, 16-bit number in ACD by taking the two's complement of the number in ACS and adding it to the number in ACD. Places the result of the addition in the shifter. Complements carry if the result is greater than 32,768. Then performs the specified shift operation and places the result of the shift in ACD if the no-load bit is 0. If the skip condition is true, skips the next sequential word.

NOTE: *If the number in ACS is less than or equal to the number in ACD, the instruction complements carry.*

Options

[c]

The processor determines the effect of carry (*c*) on the old value of carry before performing the operation (*opcode*). The following table lists the values of *c* and bits 10 and 11 and describes the operation.

Symbol *[c]*	Bits 10-11	Operation
Omitted	0 0	Leaves carry unchanged
Z	0 1	Initializes carry to 0
O	1 0	Initializes carry to 1
C	1 1	Complements carry

[sh]

The processor shifts carry 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 table lists gives the values of *sh* and bits 8 and 9 and describes the shift operation.

Symbol *[sh]*	Bits 8-9	Shift Operation
Omitted	0 0	Does not shift the result
L	0 1	Shifts left
R	1 0	Shifts right
S	1 1	Swaps 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 for testing the result of the instruction operation without destroying the destination accumulator contents. The following table lists the values of the no-load option and bit 12 and describes the operation.

Symbol *[#]*	Bit 12	Operation
Omitted	0	Loads the result into ACD
#	1	Does not load the result; restores the carry bit

NOTE: *You cannot combine the no-load option (#) with either the never skip or always skip option because the bit patterns for these combinations are used to define other types of instructions. This means that the instruction cannot end in 1000_2 or 1001_2.*

[skip]

The processor can skip the next instruction if the test condition is true. The following table lists the test conditions and the values of bits 13–15 and describes the operation.

Symbol *[skip]*	Bits 13-15	Operation
Omitted	0 0 0	Never skips
SKP	0 0 1	Always skips
SZC	0 1 0	Skips if carry is 0
SNC	0 1 1	Skips if carry is not 0
SZR	1 0 0	Skips if the result is 0
SNR	1 0 1	Skips if the result is not 0
SEZ	1 1 0	Skips if either carry or the result is 0
SBN	1 1 1	Skips if both carry and the result are not 0

When the instruction skips, it skips the next sequential 16-bit word. Be certain that a skip does not transfer control to the middle of an instruction that is 32-bits long.

SYC

System Call

SYC *acs,acd*

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

Pushes a return clock onto the stack and transfers control to the system call handler.

Disables user address translation, pushes a standard return block onto the stack, and jumps indirect through location 2. The source and destination accumulators (ACS and ACD) remain unchanged. If both accumulators are specified as AC0, no return block is pushed onto the stack and AC0 remains unchanged.

The program counter in the return block contains the address of the instruction immediately following the SYC instruction.

I/O interrupts cannot occur between the execution of the SYC instruction and the execution of the next instruction.

> **NOTE:** *DGC assemblers recognize the mnemonics SCL as equivalent to SYC1,1 and SVC as equivalent to SYC0,0.*

SZB

Skip on Zero Bit

SZB *acs,acd*

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

If the addressed bit is 0, the next sequential word is skipped.

Forms a bit pointer from the contents of ACS and ACD. ACS contains the most significant bits and ACD contains the least significant bits of the bit pointer. If ACS and ACD are specified as the same accumulator, the instruction treats the accumulator contents as the least significant 16 bits of the bit pointer and assumes the most significant 16 bits are 0.

If the addressed bit in memory is 0, the next sequential word is skipped. The contents of ACS and ACD remain unchanged.

SZBO

Skip on Zero Bit and Set to One

SZBO *acs,acd*

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

If the addressed bit is 0, sets the bit to 1 and skips the next sequential word.

Forms a bit pointer from the contents of ACS and ACD. ACS contains the most significant bits and ACD contains the least significant bits of the bit pointer. If ACS and ACD are specified as the same accumulator, the instruction treats the accumulator contents as the least significant 16 bits of the bit pointer and assumes the most significant 16 bits are 0.

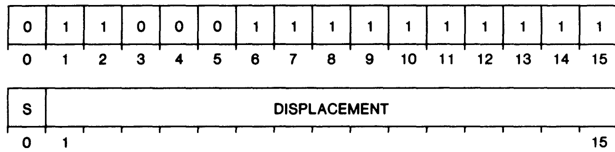
The contents of ACS and ACD remain unchanged.

> **NOTE:** *This instruction facilitates the use of bit maps for such purposes as allocating facilities (memory blocks, I/O devices, etc.) to several processes or tasks that interrupt one another or run in a multiprocessor environment.*

VCT

Vector on Interrupting Device

VCT *[@]displacement[,index]*



Returns the device code of the interrupting device and uses that code as an index into the vector table.

Depending on the mode, uses the indexed entry in the vector table in one of ways listed below:

Mode A — Uses the entry as the address of the appropriate interrupt handler.

Modes B–E — Use the entry as a pointer into a device control table (DCT), which contains the address of the appropriate interrupt handler as well as a new interrupt priority mask.

Depending on the mode, the instruction can also save the state of the computer by pushing certain information onto the stack, creating a new vector stack, setting up a priority interrupt structure, and enabling interrupts.

Three control bits determine the mode of the VCT instruction:

1. Stack change bit (S) which is bit 0 of the second word of the VCT instruction.
2. Direct bit (D) which is bit 0 of the selected vector table entry.
3. Push bit (P) which is bit 0 of the first word of the selected device control table.

The value of these bits collectively determine the mode as follows.

Direct	Stack	Push	Mode
0	—	—	A
1	0	0	B
1	0	1	C
1	1	0	D
1	1	1	E

Figure 8.5 provides an overview of the VCT instruction; Figure 8.6 presents a detailed view of the VCT instruction.

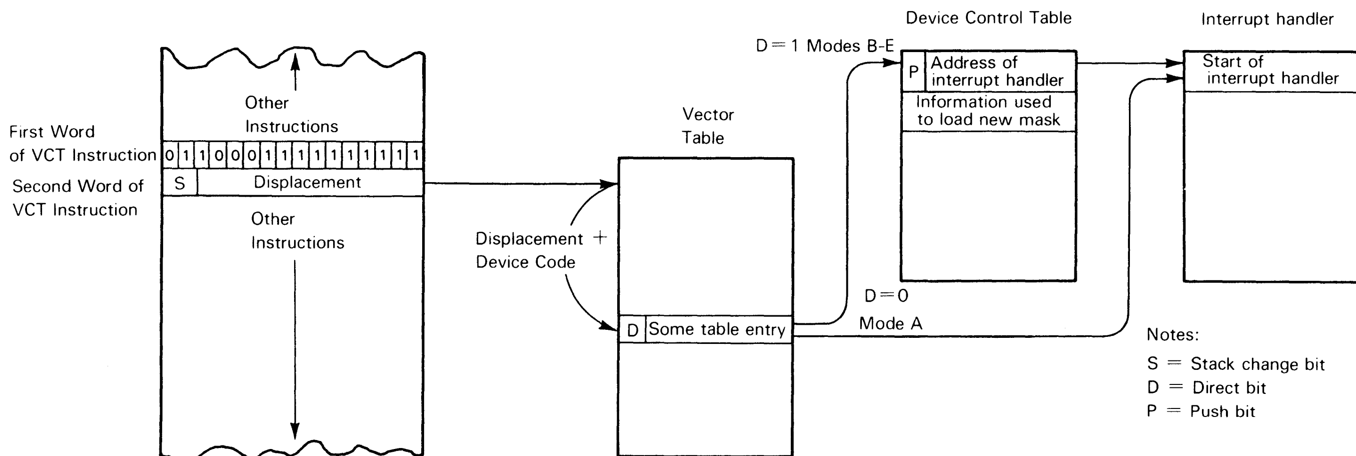


Figure 8.5 Overview of the *Vector* instruction

Common Process

All modes perform the same initital steps which begin when the instruction returns the interrupting device code. The instruction adds the device code to the address of the start of the vector table (the displacement specified by the instruction). The result is the address of an entry within the table. The instruction then fetches the contents of this vector table entry and examines bit 0 of the entry (the direct bit). If the direct bit is 0, mode A is selected; otherwise, one of the other modes (B-E) is selected, depending on the values of the stack and push bits.

Mode A

The direct bit is 0 for mode A and the values of the other control bits do not matter. In mode A, the instruction uses bits 1–15 of the fetched vector table entry as the address of the interrupt handler for the interrupting device. Control transfers immediately to this interrupt handler with all interrupts disabled.

Modes B Through E

The direct bit is 1 for modes B through E and the value of the other control bits determine which of the four modes will control the operation. The action of these modes can be divided into two parts: part 1, during which action varies from mode to mode; and part 2, during which action for all four modes is the same. In the following paragraphs, part 1 is discussed separately for each mode. The final paragraph in this section describes the common part 2.

Mode B, Part 1. Both the stack change bit and the push bit are 0 for mode B. In this mode, the instruction uses the vector table entry as the address of the device control table (DCT) for the interrupting device. Bits 1–15 of the first word of the DCT contain the address of the desired interrupt handler. The second word of the DCT contains information used to construct a new interrupt priority mask. Succeeding words (if any) contain information to be used by the device interrupt handler.

Mode C, Part 1. The stack change bit is 0 and the push bit is 1 for mode C. In this mode, the instruction performs the same functions as mode B. In addition, it pushes a standard 5-word return block onto the standard stack. The return block contains the contents of the four accumulators, the value of carry, and the contents of physical location 0 (the program counter return value).

Mode D, Part 1. The stack change bit is 1 and the push bit is 0 for mode D. In this mode, the instruction performs the same functions as mode B. In addition, it establishes a new stack for the interrupt handler (using the contents of locations 4, 6, and 7) and pushes the old contents of physical locations 40_8-43_8 (the user stack control words) onto the new stack.

Mode E, Part 1. The stack change bit and the push bit are both 1 for mode E. In this mode, the instruction combines the functions of modes C and D. That is, it performs the functions of mode B, establishes a new stack, and pushes both a 5-word return block and the old stack control words onto the new stack.

Modes B Through E, Part 2. Modes B through E perform the same procedure to complete the execution of the VCT instruction. During this procedure, the instruction pushes the current interrupt priority mask (location 5) onto the stack. Next, the instruction updates location 5 and performs the functions of a MSKO instruction, using the logical OR of the current mask and the second word of the DCT. The instruction then sets the Interrupt On (ION) flag to 1 and passes control to the selected device interrupt handler. Note that one or more instructions execute before the next I/O interrupt can occur.

Returns from VCT routines may be accomplished as follows.

Mode	Return Instruction
A	JMP @0
B	JMP @0
C	POPB
D	Restores saved stack parameters, JMP @0
E	RSTR

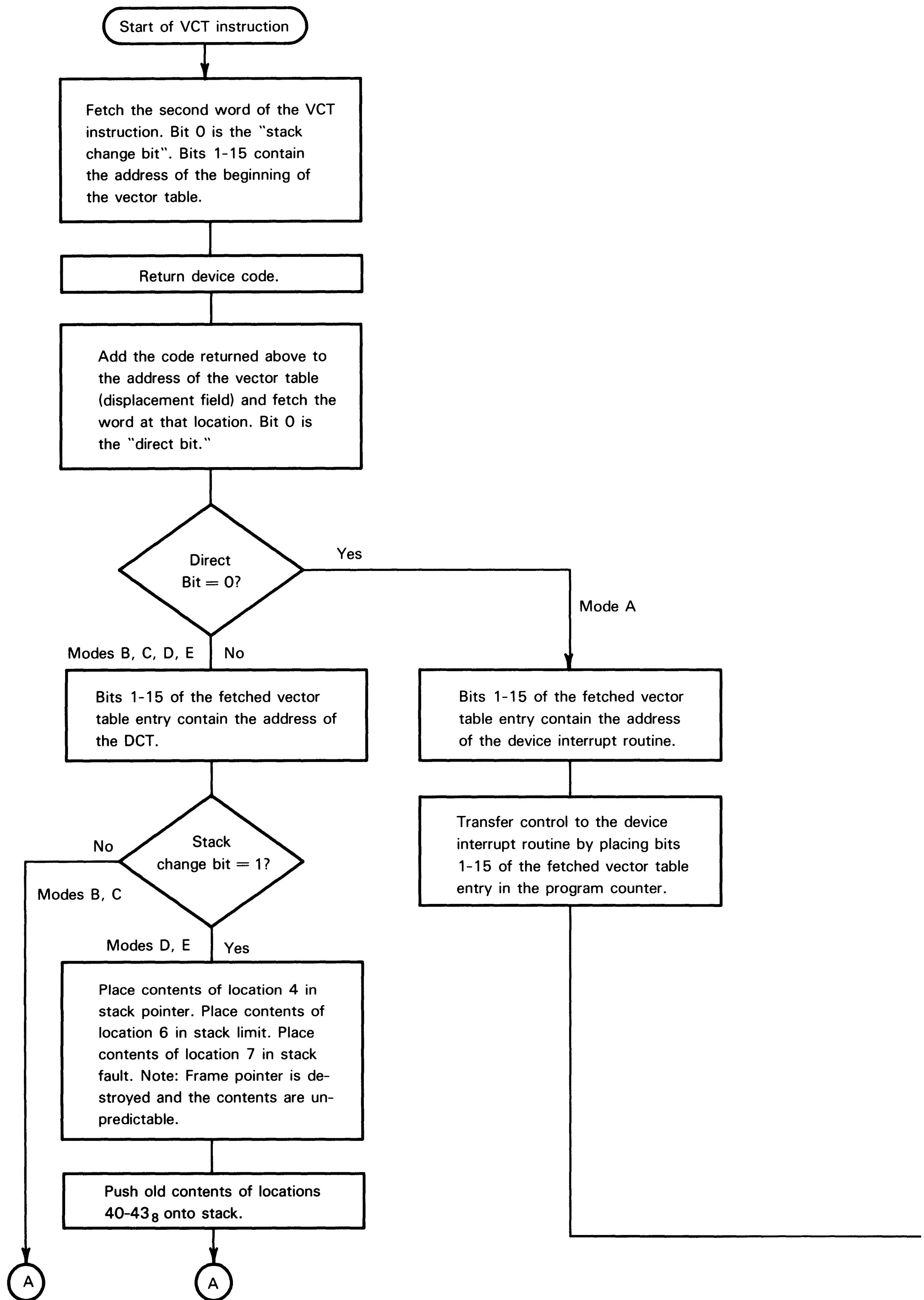
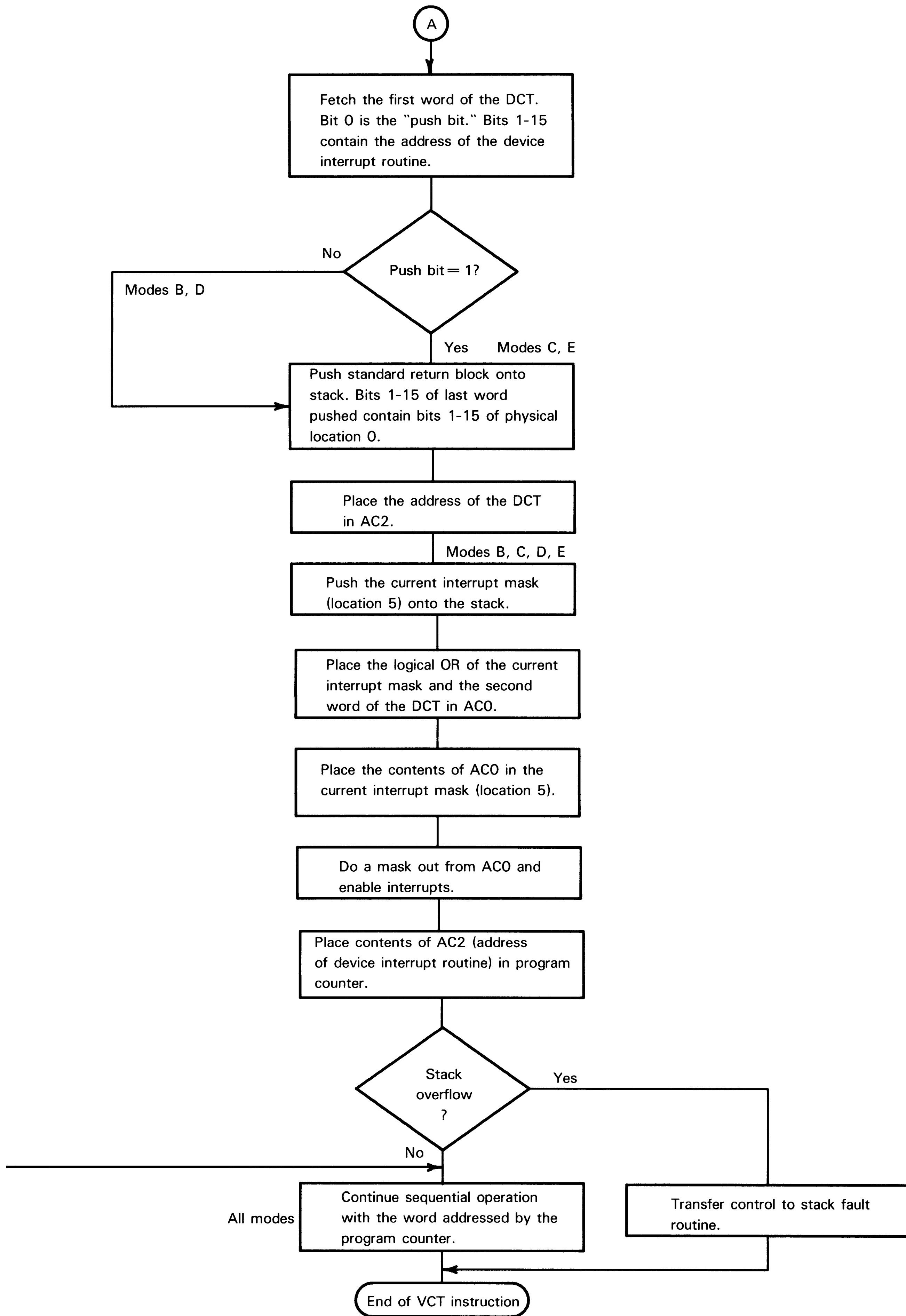


Figure 8.6 Operation of the *Vector* instruction


A
Fetch the first word of the DCT. Bit 0 is the "push bit." Bits 1-15 contain the address of the device interrupt routine.
No
Push bit = 1?
Modes B, D
Yes
Modes C, E
Push standard return block onto stack. Bits 1-15 of last word pushed contain bits 1-15 of physical location 0.
Place the address of the DCT in AC2.
Modes B, C, D, E
Push the current interrupt mask (location 5) onto the stack.
Place the logical OR of the current interrupt mask and the second word of the DCT in AC0.
Place the contents of AC0 in the current interrupt mask (location 5).
Do a mask out from AC0 and enable interrupts.
Place contents of AC2 (address of device interrupt routine) in program counter.
Stack overflow ?
Yes
No
All modes
Continue sequential operation with the word addressed by the program counter.
Transfer control to stack fault routine.
End of VCT instruction

XCH

Exchange Accumulators

XCH *acs,acd*

1	ACS		ACD		0	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

Exchanges the contents of two accumulators.

Places the original contents of ACS into ACD and the original contents of ACD in ACS.

XCT

Execute

XCT *ac*

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

Executes the contents of an accumulator as an instruction.

Executes the instruction contained in the specified accumulator as if it were in main memory in the location occupied by the *Execute* instruction.

If the instruction in the specified accumulator is an *Execute* instruction that specifies the same accumulator, the processor is placed in a one-instruction loop. Because of this possibility, the instruction is interruptible. An I/O interrupt can occur immediately prior to each execution of the instruction in the accumulator. 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.

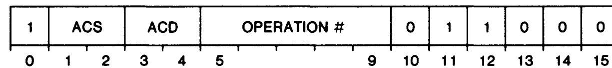
> **NOTES:** *If the specified accumulator contains the first word of a 2-word instruction, the word following the XCT instruction is used as the second word. Normal sequential operation then continues from the second word after the XCT instruction.*
>
> *Do not use the XCT instruction to execute an instruction that requires all four accumulators, such as CMV, CMT, CMP, CTR, or BAM.*

The results of XCT are undefined if the specified accumulator contains an instruction that modifies that same accumulator.

XOP

Extended Operation

XOP *acs,acd,operation #*



Pushes a return block onto the stack and transfers control to an extended operation procedure.

Pushes a standard return block onto the stack. Places the ACS's stack address in AC2 and ACD's stack address in AC3. Reserved memory location 44_8 must contain the XOP origin address which is the starting address of the 32_{10} word table of addresses. These addresses are the starting locations of the various XOP operations.

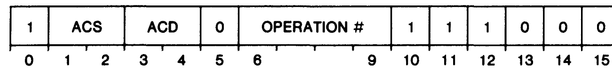
Adds the operation number in the XOP instruction to the XOP origin address to produce the address of a word in the XOP table. Fetches and treats that word as the intermediate address in the effective address calculation. After the indirection chain, if any, has been followed, the instruction places the effective address in the program counter. The contents of AC0, AC1, and the XOP origin address are unchanged.

The XOP procedure can return control to the calling program via the *Pop Block* (POPB) instruction.

XOP1

Alternate Extended Operation

XOP1 *acs,acd,operation #*

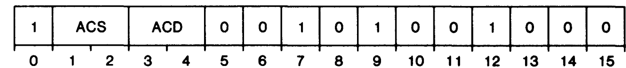


This instruction operates like the XOP instruction except that it adds 32_{10} to the operation number before it adds the operation number to the XOP origin address. In addition, the XOP1 table can contain only 16 addresses; the XOP table can contain 32 addresses.

XOR

Exclusive OR

XOR *acs,acd*



Exclusively ORs the contents of two accumulators.

Forms the logical exclusive OR of the contents of ACS and the contents of ACD and places the result in ACD. Sets a bit position in the result to 1 if the corresponding bit positions in the two operands differ; otherwise, the instruction sets the result bit to 0. The contents of ACS remain unchanged.

Example

```
XOR    0,1
```

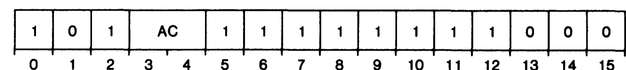
Exclusive ORs the contents of AC0 with the contents of AC1. The parameters of the operation follow.

Operation	Before	After
AC0	003156_8	023657_8
AC1	020701_8	020701_8

XORI

Exclusive OR Immediate

XORI *i,ac*



Exclusive 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 the specified accumulator and places the result in the specified accumulator.

Appendix A

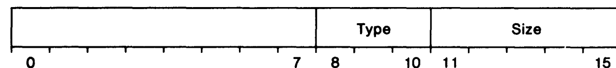
Commercial Instruction Set

The 16-bit real-time ECLIPSE (S-series) processors execute the same instructions as the 16-bit commercial ECLIPSE (C-series) processors except for the commercial instructions. Only the C-series processors execute the commercial instructions.

Data Formats

Unlike fixed-point and floating-point instructions, the commercial instructions do not imply a data format. As a result, the processor requires the program to specify an explicit data type indicator in AC1, as diagrammed below.

Explicit Data Type Indicator Format



Bits	Name	Meaning
0-7	—	Reserved for future use.
8-10	Type	Identifies type of data as categorized in Table A.1.
11-15	Size	An unsigned integer indicating the length of the data. The following list explains the data type and its corresponding size.

Data Type	Size
0 or 1	Two less than the number of decimal digits and sign.
2 or 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 data type 5, the processor expects an odd number for a size. When an even size is specified, the processor adds 1 to it (to make the size odd) and appends a 0 digit to the most significant digit.
6	The number of bytes in the two's complement number. Two bytes is the minimum size.
7	The number of bytes in the floating-point number. Four bytes is the minimum size.

An unpacked decimal string contains one ASCII character in each byte. (See Figure A.1.) Depending upon the data type and character location, the ASCII character represents a decimal digit, a sign, or a digit and a sign:

Types 0 and 1 — Combine the sign with a character. Refer to Table A.2 for a list of the sign-positioned ASCII characters. Table A.3 lists the nonsign-positioned ASCII characters.

Types 2 and 3 — Require the sign as a separate byte. The separate sign byte can be either the ASCII plus sign (+, ASCII code 053_8) or the ASCII minus sign (−, ASCII code 055_8). Table A.3 lists the nonsign-positioned ASCII characters.

A packed decimal string contains two BCD digits per byte. (See Figure A.1.) The most significant digit contains a 0 if the decimal string contains an odd number of digits. The last byte must contain the least significant digit and the sign. 15_8 (or D_{16}) represents the minus sign (−); 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	3 (063) 9 (071) P (120)	(000002)
		+397	3 (063) 9 (071) G (107)	
1	*Unpacked decimal* — first byte combines the sign and the first digit	−397	L (114) 9 (071) 7 (067)	(000042)
		+397	C (103) 9 (071) 7 (067)	
2	*Unpacked decimal* — last byte contains the sign	−397	3 (063) 9 (071) 7 (067) - (055)	(000103)
		+397	3 (063) 9 (071) 7 (067) + (070)	
3	*Unpacked decimal* — first byte contains the sign	−397	- (055) 3 (063) 9 (071) 7 (067)	(000143)
		+397	+ (070) 3 (063) 9 (071) 7 (067)	
4	*Unpacked decimal* — and unsigned	397	3 (063) 9 (071) 7 (067)	(000202)
5	*Packed decimal* — and BCD digits (or sign) per byte	-397	39 (071) 7 - (175) 39 [39] 7 - [7D]	(000243)
		+397	39 (071) 7+ (177) 39 [39] 7+ [7F]	
6	*Two's complement* — byte-aligned	−397	(-615) = (176) (163) = (177163)	(000302)
		+397	(+615) = (001) (215) = (000615)	
7	*Floating point* — byte-aligned	−397	(06771630000) [37] [E7] [30] [00]	(000344)
		+397	(06706150000) [37] [18] [D0] [00]	

Table A.1 Explicit data types

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

Table A.2 Sign and number combination for unpacked decimal

[1] *Octal code*

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

Table A.3 Nonsign-positioned numbers for unpacked decimal

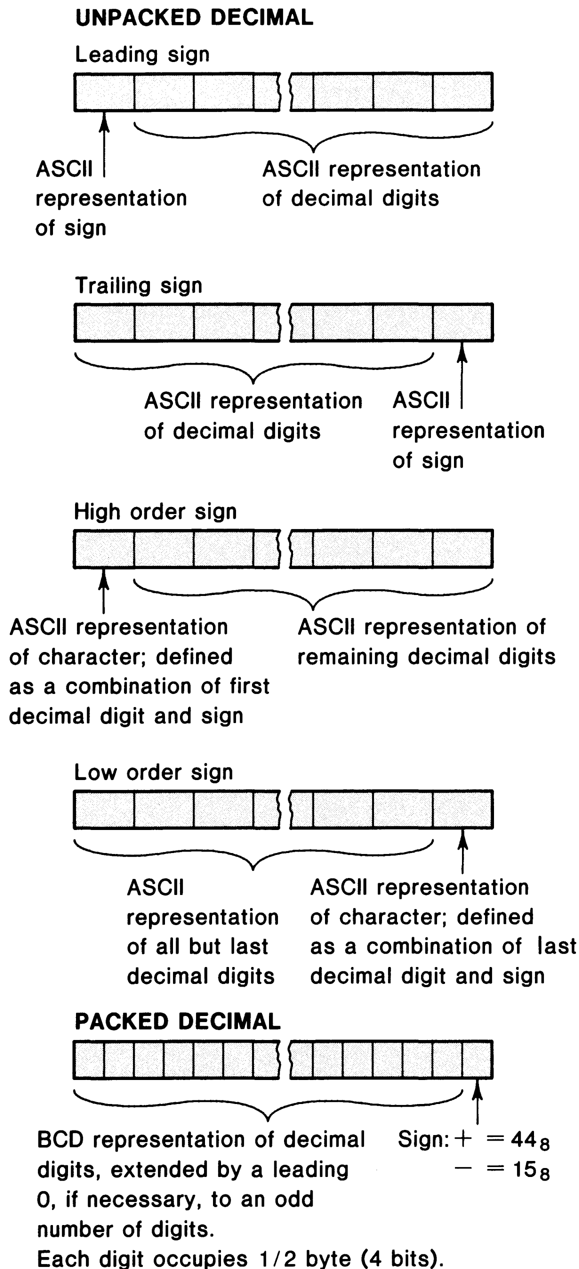


Figure A.1 Packed and unpacked decimal data

Move and Sign Instructions

Commercial instructions allow the program to

- Convert and insert a string of decimal or ASCII characters;
- Evaluate the sign of a decimal number;
- 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 A.4 lists these instructions. Detailed descriptions of each instruction follow.

Mnemonic	Name	Action
EDIT	*Edit*	Converts a decimal integer to a string of bytes, moves a string of bytes, or inserts additional bytes under the control of an edit subprogram. Table A.5 lists the edit subprogram instructions.
LDI	*Load Integer*	Translates a decimal integer in memory to floating-point format and places the result in a floating-point accumulator.
LDIX	*Load Integer Extended*	Distributes a decimal integer of data type 0 - 5 into the four floating-point accumulators.
LSN	*Load Sign*	Identifies the sign of a decimal number.
STI	*Store Integer*	Converts the contents of a floating-point accumulator to a specified format and places it in memory.
STIX	*Store Integer Extended*	Converts the contents of four floating-point accumulators to an integer of data type 0 - 5 and uses the least significant eight digits of each to form a 32-digit integer.

Table A.4 Commercial instructions

Mnemonic	Name	Action
DADI	*Add to DI*	Adds a signed integer to the destination indicator.
DAPS	*Add to P Depending on S*	Adds a signed integer to the opcode pointer if the Sign flag is 0.
DAPT	*Add to P Depending on T*	Adds a signed integer to the opcode pointer if the significance Trigger is 1.
DAPU	*Add to P*	Adds a signed integer to the opcode pointer.
DASI	*Add to SI*	Adds a signed integer to the source indicator.
DDTK	*Decrement and Jump if Nonzero*	Decrements a word in the stack by one and jumps if the word is nonzero.
DEND	*End Edit*	Ends the edit subprogram.
DICI	*Insert Characters Immediate*	Inserts characters into a string.
DIMC	*Insert Character J Times*	Inserts a character into a string a specified number of times.
DINC	*Insert Character Once*	Inserts a character into a string once.
DINS	*Insert Sign*	Inserts character-a or character-b depending on the Sign flag.
DINT	*Insert Character Suppress*	Insert character-a or character-b depending on the significance Trigger.
DMVA	*Move Alphabetics*	Moves a specified number of alphabetic characters.
DMVC	*Move Characters*	Moves a specified number of characters.
DMVF	*Move Float*	Moves a specified number of floating-point numbers.
DMVN	*Move Numerics*	Moves a specified number of numerics.
DMVO	*Move Digit with Overpunch*	Moves a digit plus overpunch.
DMVS	*Move Numeric with Zero Suppress*	Moves numerics and suppresses zeros.
DNDF	*End Float*	Ends floating-point.
DSSO	*Set Sign to One*	Sets Sign flag to 1.
DSSZ	*Set S to Zero*	Sets Sign flag to 0.
DSTK	*Store in Stack*	Stores a byte on the stack.
DSTO	*Set T to One*	Sets the significance Trigger to 1.
DSTZ	*Set T to Zero*	Sets the significance Trigger to 0.

Table A.5 Edit subprogram instructions

EDIT

Edit

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

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. If data type 4 is specified, the instruction also performs operations on alphanumeric data.

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 AC3. DI is set to the value contained in AC2, and P is set to the value contained in AC0. At this time, the sign of the source number is also checked for validity.

The subprogram consists 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 otherwise, after each operation the *Edit* instruction updates P 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 modify *S*, *T*, SI, DI and P, and also test *S* and *T*.

The accumulators define the *Edit* operation as follows:

AC0 contains a byte pointer to the first opcode of the *Edit* subprogram.

AC1 contains the data-type indicator describing the number to be processed.

AC2 contains a byte pointer to the the first byte of the destination field.

AC3 contains 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 certain types of overlap may produce unusual side effects.

Upon successful completion, the carry and accumulators are as follows:

Carry contains the significance Trigger (*T*).

AC0 contains a byte pointer (P) to the next opcode to be processed.

AC1 is undefined.

AC2 contains a byte pointer (DI) to the next destination byte.

AC3 contains a byte pointer (SI) to the next source byte.

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 move characters from one string to another. For those opcodes which move numeric data, special actions may be performed. For others which move non-numeric data, characters are copied literally to the destination.

The *Edit* instruction places information on the stack. Therefore, the stack must be set up and contain at least nine words available for use.

If the *Edit* instruction is interrupted, it places restart information on the stack and places 177777_8 in AC0. If the initial contents of AC0 equal 177777_8, the *Edit* instruction assumes it is restarting from an interrupt; therefore, the program should not allow this to occur under any other circumstances.

In the description of some of the *Edit* opcodes, the symbol *j* denotes how many characters a certain operation should process. When its most significant bit is 1, *j*'s meaning is different: it points to a word in the stack that denotes the number of characters the instruction should process. So, in those cases where the most significant 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*.

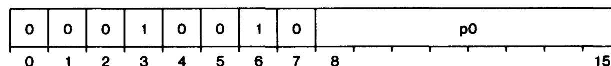
The operation uses the number at this address as a character count instead of *j*.

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.

DADI

Add to DI

DADI *p0*

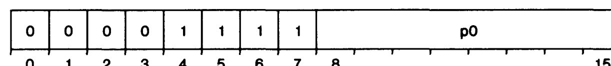


Adds the 8-bit two's complement integer specified by *p0* to the Destination Indicator (DI).

DAPS

Add to P Depending on *S*

DAPS *p0*

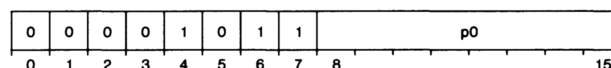


If *S* is 0, the instruction adds the 8-bit two's complement integer specified by *p0* to the opcode Pointer (P). Before the add is performed, P points to the byte containing the DAPS opcode.

DAPT

Add to P Depending on *T*

DAPT *p0*

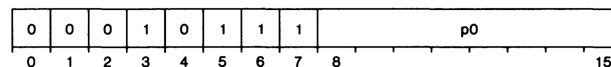


If *T* is 1, the instruction adds the 8-bit two's complement integer specified by *p0* to the opcode Pointer (P). Before the add is performed, P points to the byte containing the DAPT opcode.

DAPU

Add to P

DAPU *p0*

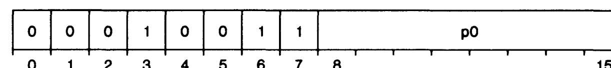


Adds the 8-bit two's complement integer specified by *p0* to the opcode Pointer (P). Before the add is performed, P points to the byte containing the DAPU opcode.

DASI

Add to SI

DASI *p0*

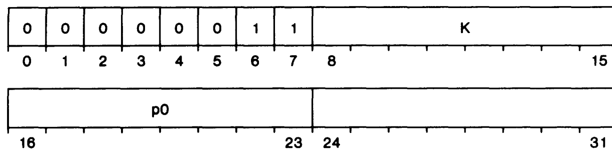


Adds the 8-bit two's complement integer specified by *p0* to the Source Indicator (SI).

DDTK

Decrement and Jump if Nonzero

DDTK *k,p0*



Decrements a word in the stack by 1. 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 points to the byte containing the DDTK opcode.

If the 8-bit two's complement integer specified by *k* is negative, the word decremented is at the address,

stack pointer+1+*k*

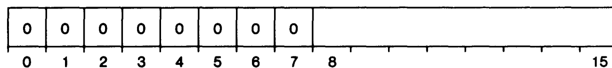
If *k* is positive, the word decremented is at the address,

frame pointer+1+*k*.

DEND

End Edit

DEND

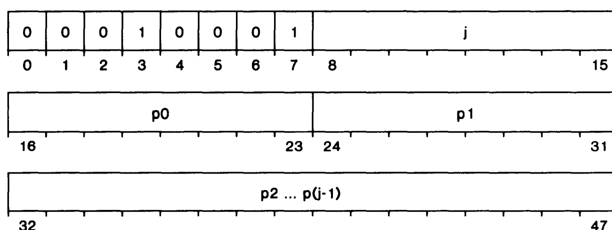


Terminates the *Edit* subprogram.

DICI

Insert Characters Immediate

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

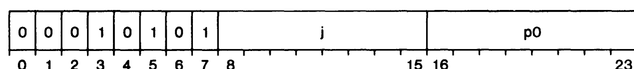


Inserts *j* characters from the opcode stream into the destination field beginning at the position specified by DI. Increases P by *j*+2 and increases DI by *j*.

DIMC

Insert Character *j* Times

DIMC *j,p0*

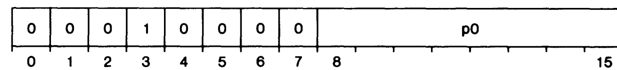


Inserts the character specified by *p0* into the destination field a number of times equal to *j*, beginning at the position specified by DI. Increases DI by *j*.

DINC

Insert Character Once

DINC *p0*

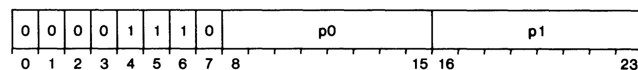


Inserts the character specified by *p0* in the destination field at the position specified by DI. Increments DI by 1.

DINS

Insert Sign

DINS *p0,p1*

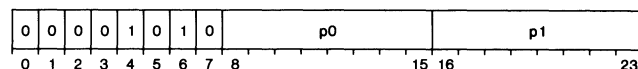


If the Sign flag (*S*) is 0, the instruction inserts the character specified by *p0* in the destination field at the position specified by DI. If *S* is 1, the instruction inserts the character specified by *p1* in the destination field at the position specified by DI. Increments DI by 1.

DINT

Insert Character Suppress

DINT *p0,p1*

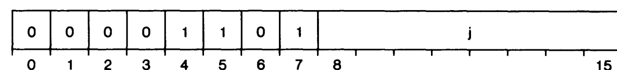


If the significance Trigger (*T*) is 0, the instruction inserts the character specified by *p0* in the destination field at the position specified by DI. If *T* is 1, the instruction inserts the character specified by *p1* in the destination field at the position specified by DI. Increments DI by 1.

DMVA

Move Alphabetics

DMVA *j*



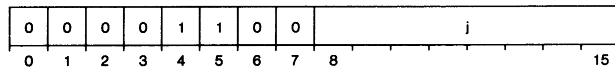
Moves *j* characters from the source field (beginning at the position specified by SI) to the destination field (beginning at the position specified by DI). Increases both SI and DI by *j*. Sets *T* to 1.

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. (Alphabetic characters are A–Z, a–z, or space).

DMVC

Move Characters

DMVC *j*



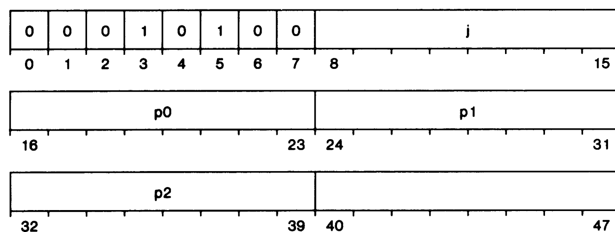
Increments SI if the source data type is 3 and $j>0$. The instruction then moves *j* characters from the source field (beginning at the position specified by SI) to the destination field (beginning at the position specified by DI). Increases SI and DI by *j*. Sets *T* to 1.

Initiates a commercial fault if the attribute specifier word indicates that the source is data type 5 (packed). Does not check the validity of the characters.

DMFV

Move Float

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 instruction places both the digit substitute and the digit in the destination field. SI increases by *j* and DI increases by $j+1$. When *T* does not change from 0 or 1, 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.

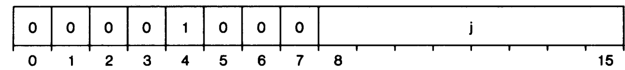
If the source data type is 2, the state of SI is undefined after the least significant digit has been processed.

The instruction initiates a commercial fault if any of the digits processed is not valid for the specified data type.

DMVN

Move Numerics

DMVN *j*



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 both SI and DI by *j* and sets *T* to 1.

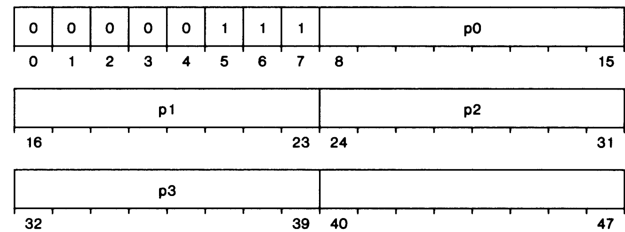
For data type 2, the state of SI is undefined after the least significant digit has been processed.

Initiates a commercial fault if any of the characters moved is not valid for the specified data type.

DMVO

Move Digit with Overpunch

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). Increases both SI and DI by 1.

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

Table A.6 summarizes the effects of DMVO.

Values		
Digit	***S***	**Action**
0 or space	0	Places *p0* in destination field
.	1	Places *p1* in destination field
≠0	0	Adds *p2* to the digit and places the result in the destination field[1]
	1	Adds *p3* to the digit and places the result in the destination field[1]
		Sets *T* to 1

Table A.6 Results of DMVO

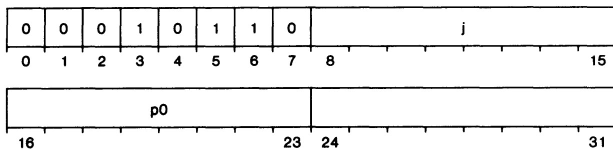
[1] *The instruction assumes that* p2 *and* p3 *are ASCII characters.*

The instruction initiates a commercial fault if the character is not valid for the specified data type.

DMVS

Move Numeric with Zero Suppression

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. Increases both SI and DI by *j*.

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

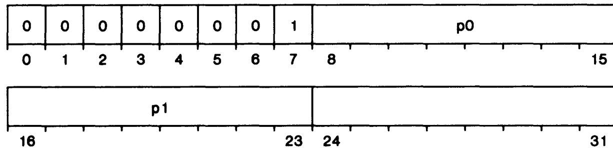
This opcode destroys the data type specifier.

Initiates a commercial fault if any of the characters moved is not numeric (0–9) or space.

DNDF

End Float

DNDF *p0,p1*

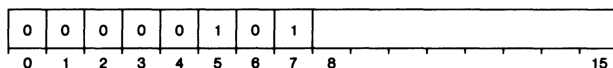


If *T* is 1, the instruction places nothing in the destination field and leaves DI unchanged. If *T* and *S* are both 0, the instruction places *p0* in the destination field at the position specified by DI. If *T* is 0 and *S* is 1, the instruction places *p1* in the destination field at the position specified by DI. Increases DI by 1 and sets *T* to 1.

DSSO

Set *S* to One

DSSO

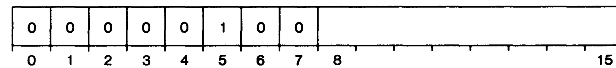


Sets the Sign flag (*S*) to 1.

DSSZ

Set *S* to Zero

DSSZ

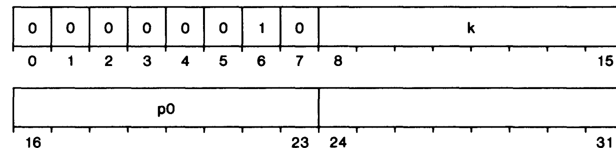


Sets the Sign flag (*S*) to 0.

DSTK

Store In Stack

DSTK *k,p0*



Stores the byte specified by *p0* in bits 8–15 of a word in the 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

stack pointer+1+*k*.

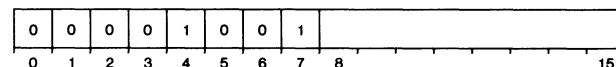
If *k* is positive, then the instruction stores *p0* at the address,

frame pointer+1+*k*.

DSTO

Set *T* to One

DSTO

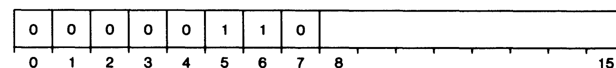


Sets the significance Trigger (*T*) to 1.

DSTZ

Set *T* to Zero

DSTZ

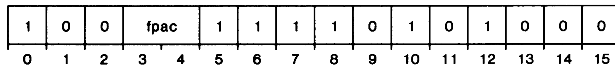


Sets the significance Trigger (*T*) to 0.

LDI

Load Integer

LDI *fpac*



Translates a string of bytes 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 string of bytes to floating-point form, normalizes it, and places it in the specified FPAC. The instruction updates the Zero (Z) and Negative (N) flags in the floating-point status register to reflect the new contents of the specified FPAC. Leaves the decimal number unchanged in memory and destroys the previous contents of the specified FPAC.

Two accumulators define the operation as follows:

- AC1 contains the data-type indicator describing the number.
- AC3 contains 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 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 of the load operation, the four accumulators and carry are as follows:

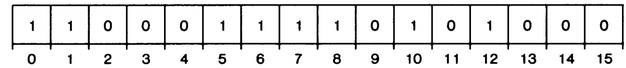
- AC0 and AC1 are unchanged.
- AC2 contains the original contents of AC3.
- AC3 is undefined.
- Carry is unchanged.

NOTE: *An attempt to load a minus 0 sets the specified FPAC to true zero.*

LDIX

Load Integer Extended

LDIX



Distributes a decimal integer of data type 0, 1, 2, 3, 4, or 5 into the four floating-point accumulators (FPACs).

Extends the integer with high-order zeros until it is 32 digits long. Divides the integer into four units of eight digits each and converts each unit to a floating-point number. Places the number obtained from the eight high-order digits into FPAC0; the number otained by the next eight digits into FPAC1; the number obtained from the next eight digits into FPAC2; and the number obtained from the eight low-order digits into FPAC3. The instruction places the sign of the integer in each FPAC unless that FPAC has received eight digits of zeros, in which case the instruction sets the FPAC to true zero. The Zero (Z) and Negative (N) flags in the floating-point status register are unpredictable.

Two accumulators define the operation as follows:

- AC1 contains the data type indicator describing the integer.
- AC3 contains a byte-pointer which is the address of the high-order byte of the integer.

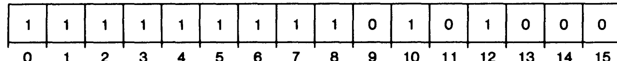
Upon successful completion of the operation, the four accumulators and carry are as follows:

- AC0 and AC3 are undefined.
- AC1 is unchanged.
- AC2 contains the original contents of AC3.
- Carry is unchanged.

LSN

Load Sign

LSN



Under 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 instruction leaves the number in memory unchanged. The meaning of the return code is as follows:

Value of Number	Code
Positive nonzero	+1
Negative nonzero	−1
Positive zero	0
Negative zero	−2

Two accumulators define the operation as follows:

- AC1 contains the data type indicator describing the number.
- AC3 contains a byte pointer which is the address of the high-order byte of the number.

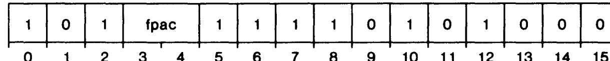
Upon successful completion of the operation, the accumulators and carry are as follows:

- AC0 is unchanged.
- AC1 contains the value code.
- AC2 contains the original contents of AC3.
- AC3 is unpredictable.
- Carry is unchanged.

STI

Store Integer

STI *fpac*



Converts the contents of a floating-point accumulator (FPAC) 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 the specified data 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). Two accumulators define the operation as follows:

- AC1 contains the data type indicator describing the integer.
- AC3 contains a byte-pointer which is the address of the high-order byte of the destination field in memory.

After successful completion of the operation, the accumulators are as follows:

- AC0 and AC1 are unchanged.
- AC2 contains the original contents of AC3.
- AC3 contains a byte pointer which is the address of the next byte after the destination field.

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 destination field is not large enough to contain the number being stored, the instruction disregards high-order digits until the number will fit. The instruction stores the 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.

STIX

Store Integer Extended

STIX

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

Converts the contents of four floating-point accumulators (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 decimal 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, and the condition codes in the floating-point status register are unpredictable.

Two accumulators define the operation as follows:

- AC1 contains the data type indicator describing the integer.
- AC3 contains a byte-pointer which is the address of the high-order byte of the destination field in memory.

After successful completion of the operation, the accumulators are as follows:

- AC0 is undefined.
- AC1 is unchanged.
- AC2 contains the original contents of AC3.
- AC3 contains a byte pointer which is the address of the next byte after the destination field.

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 destination field is not large enough to contain the number being stored, then for data types 0 through 6, the high-order digits are discarded until the number fits into the field. The remaining low-order digits are stored and carry is set to 1. For data type 7, the low-order digits are discarded.

If the number being stored will not fill the destination field, then for data types 0 through 5, the high-order bytes are set to 0 until the number fits. For data type 6, the sign bit is extended to the left to fill the field. For data type 7, the low-order bytes are set to 0.

Index

Within the index, the letter "f" following a page number indicates "and the following page or pages"; a boldface page number identifies a dictionary entry.

A

Abbreviations i
Absolute Value (FAB) instruction **74**
Absolute, addressing 6
Access
  bit 8
  byte 7f
  device 35f
  word 7
Accumulator formats 3
Accumulator-relative addressing 6
Add (ADD) instruction **48f**
Add Complement (ADC) instruction **47f**
Add Double, FPAC to FPAC (FAD) instruction **74**
Add Double, Memory to FPAC (FAMD) instruction **75**
Add Immediate (ADI) instruction **50**
Add Single, FPAC to FPAC (FAS) instruction **75**
Add Single, Memory to FPAC (FAMS) instruction **75**
Add to DI (DADI) instruction **119**
Add to P (DAPU) instruction **119**
Add to P Depending on S (DAPS) instruction **119**
Add to P Depending on T (DAPT) instruction **119**
Add to SI (DASI) instruction **119**
Addition
  instruction lists
    fixed-point 16
    floating-point 21
  operations, floating-point 21
Address translation. *See* Address translator.
Address translator
  fault handling 43f
  instruction lists 44
  map table 41f
  memory management 4
  page 31 register 42
  protection 43
  status 44
Address, effective 6f
Addressing
  absolute 6
  accumulator-relative 6
  bit 8
  byte 7f
  indirect 7
  logical 4
  lower page zero 6f
  modes 6f
  physical 4
  program-counter-relative 6
  translation 41f
  word 7
Alternate Extended Operation (XOP1) instruction **113**
AND (AND) instruction **51f**
AND Immediate (ANDI) instruction **52**
AND with Complemented Source (ANC) instruction **51**
Arithmetic
  data formats
    fixed-point 13
    floating-point 19
  instruction lists
    fixed-point 16
    floating-point 21f

B

Binary coded decimal. *See* Decimal.
Bit
  accessing 8
  addressing 8
  pointer format 8
Block Add and Move (BAM) instruction **53**
Block Move (BLM) instruction **54**
Burst multiplexor channel (BMC) 4, 41f
BMC address translation
  instruction list 44
  map table 41f
Byte
  accessing 7f
  addressing 7f
  data format 14
  instruction list 18
  pointer format 7

C

Carry
- instruction list 16
- operations 14

Character Compare (CMP) instruction **56**
Character Move (CMV) instruction **58**
Character Move Until True (CMT) instruction **57**
Character Translate (CRT) instruction **60f**
Clear Errors (FCLE) instruction **76**
Commercial instructions 115f
- data formats and type indicators 115f
- edit subprogram instructions 118f
- instruction lists 117f

Compare Floating-Point (FCMP) instruction **76**
Compare to Limits (CLM) instruction **55**
Complement (COM) instruction **59f**
Computation
- fixed-point 3, 13f
- floating-point 3, 19f

Conventions i
Count Bits (COB) instruction **59**

D

Data channel 4
Data conversion, instruction list 20
Data formats,
- byte 14
- commercial 115f
- decimal 14
- fixed-point 13f
- floating-point 19
- logical 13f

Data in A (DIA) instruction **63**
Data in B (DIB) instruction **64**
Data in C (DIC) instruction **64**
Data movement instruction lists
- fixed-point 15
- floating-point 21

Data Out A (DOA) instruction **68**
Data Out B (DOB) instruction **68**
Data Out C (DOC) instruction **69**
Data type indicator 115f
Decimal Add (DAD) instruction **62**
Decimal Subtract (DSB) instruction **69**
Decimal
- data formats 14
- instruction lists 18
- packed 116f
- unpacked 116f

Decrement and Jump if Nonzero (DDTK) instruction **120**
Decrement and Skip if Zero (DSZ) instruction **71**
Device
- access 35f
- flags 36f
- management 4, 35f

Disable User Mode (NIOP MAP) instruction **100**
Disable User Translation (NIOP MAP) instruction **100**
Dispatch (DSPA) instruction **70**
Displacement, memory reference 5
Divide Double, FPAC by FPAC (FDD) instruction **76**
Divide Double, FPAC by Memory (FDMD) instruction **76**
Divide Single, FPAC by FPAC (FDS) instruction **77**
Divide Single, FPAC by Memory (FDMS) instruction **77**
Division
- instruction lists
  - fixed-point 16
  - floating-point 22
- operation, floating-point 22

Double Hex Shift Left (DHXL) instruction **62**
Double Hex Shift Right (DHXR) instruction **63**
Double Logical Shift (DLSH) instruction **67**
Double word access 7
Double-precision data formats
- fixed-point 13
- floating-point 19

E

Edit (EDIT) instruction **118f**
Edit subprogram instructions 118f
Effective address 6f
Enable ERCC (DOA ERCC) instruction **68**
End Edit (DEND) instruction **120**
End Float (DNDF) instruction **122**
ERCC. *See* Error checking and correction.
Error checking and correction
- instruction list 45
- memory 4

Exchange Accumulators (XCH) instruction **112**
Exclusive OR (XOR) instruction **113**
Exclusive OR Immediate (XORI) instruction **113**
Execute (XCT) instruction 25, **112**
Extended Add Immediate (ADDI) instruction **50**
Extended Decrement and Skip if Zero (EDSZ) instruction **71**
Extended Increment and Skip if Zero (EISZ) instruction **71**
Extended Jump (EJMP) instruction **71**
Extended Jump to Subroutine (EJSR) instruction **71**
Extended Load Accumulator (ELDA) instruction **72**
Extended Load Byte (ELDB) instruction **72**
Extended Load Effective Address (ELEF) instruction **73**
Extended Operation (XOP) instruction **113**
Extended Store Accumulator (ESTA) instruction **73**
Extended Store Byte (ESTB) instruction **4**

F

Faults
- floating-point 23, 33
- handling 31f
- stack 12, 32f

Fix to AC (FFAS) instruction **78**
Fix to Memory (FFMD) instruction **78**
Fixed-point computation 13f
accumulator formats 3
common operations 14f
shift 14f
skip 15
data formats 13f
Float from AC (FLAS) instruction **79**
Float from Memory (FLMD) instruction **80**
Floating-point computation 19f
accumulators formats 3
data formats 19
fault handling 33
faults 23
instruction lists 23
operations 19f
addition 21
calculation 20
data storage 20
division 22
guard digits 20
mantissa alignment 20
multiplication 21f
normalization 20
subtraction 21
truncation 20
status 23
status register format 23

G

Guard digits 20

H

Halt (HALT) instruction **88**
Halve (FHLV) instruction **79**
Halve (HLV) instruction **88**
Hex Shift Left (HXL) instruction **89**
Hex Shift Right (HXR) instruction **89**

I

I/O Protect flag 35
I/O Reset (IORST) instruction **92**
I/O Skip (SKP) instruction **105**
I/O. *See* Input/output.
Inclusive OR (IOR) instruction **91**
Inclusive OR Immediate (IORI) instruction **91**
Increment (INC) instruction **90**
Increment and Skip if Zero (ISZ) instruction **92**
Indexing 5
Indirect addressing 7
Indirection protection 43
Initiate Page Check (DOC MAP) instruction **69**
Input/output
facilities 4
general instructions
device flags 36f
format 36
instruction list 36
protection 43
system 2
Insert Character Once (DINC) instruction **120**
Insert Character Suppress (DINT) instruction **120**
Insert Characters Immediate (DICI) instruction **120**
Insert Characters j Times (DIMC) instruction **120**
Insert Sign (DINS) instruction **120**
Instruction interruption 37

I

Instruction lists
address translator 44
commercial 117f
error checking and correction 45
fixed-point
addition 16
byte 18
data movement 15
decimal 18
division 16
initialize carry 16
logical 17f
multiplication 16
shift 17f
skip 17f
subtraction 16
floating-point
addition 21
data conversion 20
data movement 21
division 22
multiplication 22
subtraction 21
general I/O 36
interrupt system 37
jump 25
stack 10f
subroutine 27f

I

Instruction fields
accumulator (AC, FPAC) 5
displacement 5
index 5
indirect (@) 5
op code 5
Instructions
Absolute Value (FAB) **74**
Add (ADD) **48f**
Add Complement (ADC) **47f**
Add Double, FPAC to FPAC (FAD) **74**

Instructions
Add Double, Memory to FPAC (FAMD) **75**
Add Immediate (ADI) **50**
Add Single, FPAC to FPAC (FAS) **75**
Add Single, Memory to FPAC (FAMS) **75**
Add to DI (DADI) **119**
Add to P (DAPU) **119**
Add to P Depending on S (DAPS) **119**
Add to P Depending on T (DAPT) **119**
Add to SI (DASI) **119**
Alternate Extended Operation (XOP1) **113**
AND (AND) **51f**
AND Immediate (ANDI) **52**
AND with Complemented Source (ANC) **51**
Block Add and Move (BAM) **53**
Block Move (BLM) **54**
Character Compare (CMP) **56**
Character Move (CMV) **58**
Character Move Until True (CMT) **57**
Character Translate (CRT) **60f**
Clear Errors (FCLE) **76**
Compare Floating-Point (FCMP) **76**
Compare to Limits (CLM) **55**
Complement (COM) **59f**
Count Bits (COB) **59**
Data in A (DIA) **63**
Data in B (DIB) **64**
Data in C (DIC) **64**
Data Out A (DOA) **68**
Data Out B (DOB) **68**
Data Out C (DOC) **69**
Decimal Add (DAD) **62**
Decimal Subtract (DSB) **69**
Decrement and Jump if Nonzero (DDTK) **120**
Decrement and Skip if Zero (DSZ) **71**
Disable User Mode (NIOP MAP) **100**
Disable User Translation (NIOP MAP) **100**
Dispatch (DSPA) **70**
Divide Double, FPAC by FPAC (FDD) **76**
Divide Double, FPAC by Memory (FDMD) **76**
Divide Single, FPAC by FPAC (FDS) **77**
Divide Single, FPAC by Memory (FDMS) **77**
Double Hex Shift Left (DHXL) **62**
Double Hex Shift Right (DHXR) **63**
Double Logical Shift (DLSH) **67**
Edit (EDIT) **118f**
Enable ERCC (DOA ERCC) **68**
End Edit (DEND) **120**
End Float (DNDF) **122**
Exchange Accumulators (XCH) **112**

Instructions
Exclusive OR (XOR) **113**
Exclusive OR Immediate (XORI) **113**
Execute (XCT) **112**
Extended Add Immediate (ADDI) **50**
Extended Decrement and Skip if Zero (EDSZ) **71**
Extended Increment and Skip if Zero (EISZ) **71**
Extended Jump (EJMP) **71**
Extended Jump to Subroutine (EJSR) **71**
Extended Load Accumulator (ELDA) **72**
Extended Load Byte (ELDB) **72**
Extended Load Effective Address (ELEF) **73**
Extended Operation (XOP) **113**
Extended Store Accumulator (ESTA) **73**
Extended Store Byte (ESTB) **74**
Fix to AC (FFAS) **78**
Fix to Memory (FFMD) **78**
Float from AC (FLAS) **79**
Float from Memory (FLMD) **80**
Halt (HALT) **88**
Halve (FHLV) **79**
Halve (HLV) **88**
Hex Shift Left (HXL) **89**
Hex Shift Right (HXR) **89**
I/O Reset (IORST) **92**
I/O Skip (SKP) **105**
Inclusive OR (IOR) **91**
Inclusive OR Immediate (IORI) **91**
Increment (INC) **90**
Increment and Skip if Zero (ISZ) **92**
Initiate Page Check (DOC MAP) **69**
Insert Character Once (DINC) **120**
Insert Character Suppress (DINT) **120**
Insert Characters Immediate (DICI) **120**
Insert Characters j Times (DIMC) **120**
Insert Sign (DINS) **120**
Integerize (FINT) **79**
Interrupt Acknowledge (INTA) **91**
Interrupt Disable (INTDS) **91**
Interrupt Enable (INTEN) **91**
Jump (JMP) **92**
Jump to Subroutine (JSR) 28f, **92**
Load Accumulator (LDA) **93**
Load Byte (LDB) **93**
Load Effective Address (LEF) **93**
Load Exponent (FEXP) **77**
Load Floating-Point Double (FLDD) **79**
Load Floating-Point Single (FLDS) **80**
Load Floating-Point Status (FLST) **80**
Load Integer (LDI) **123**
Load Integer Extended (LDIX) **123**
Load Map (LMP) **94**
Load Sign (LSN) **124**
Load User/DCH Map Table (LMP) **94**

Instructions
Load User/DCH Translator Status (DOA MAP) **68**
Locate and Reset Lead Bit (LRB) **94**
Locate Lead Bit (LOB) **94**
Logical Shift (LSH)) **94**
Map Single Cycle (NIOP MAP) **100**
Map Supervisor Page 31 (DOB MAP) **68**
Mask Out (MSKO) **96**
Modify Stack Pointer (MSP) **96**
Move (MOV) **95**
Move Alphabetics (DMVA) **120**
Move Characters (DMVC) **121**
Move Digit with Overpunch (DMVO) **121**
Move Float (DMFV) **121**
Move Floating-Point (FMOV) **81**
Move Numeric with Zero Suppression (DMVS) **122**
Move Numerics (DMVN) **121**
Multiply Double, FPAC by FPAC (FMD) **80**
Multiply Double, FPAC by Memory (FMMD) **81**
Multiply Single, FPAC by FPAC (FMS) **81**
Multiply Single, FPAC by Memory (FMMS) **81**
Negate (FNEG) **82**
Negate (NEG) **99**
No I/O Transfer (NIO) **100**
Normalize (FNOM) **82**
No Skip (FNS) **82**
Page Check (DIC MAP) **64**
Pop Block (POPB) **101**
Pop Floating-Point State (FPOP) **82**
Pop Multiple Accumulators (POP) **100**
Pop PC and Jump (POPJ) **101**
Push Floating-Point State (FPSH) **83**
Push Jump (PSHJ) **101**
Push Multiple Accumulators (FPSH) **101**
Push Return Address (PSHR) **102**
Read BMC Status (DIB BMC) **64**
Read High Word (FRH) **83**
Read Memory Fault Address (DIA ERCC) **63**
Read Memory Fault Code and Address (DIB ERCC) **64**
Read Switch Register (READS) **102**
Read User/DCH Translator Status (DIA MAP) **63**
Restore (RTSR) **102**
Return (RTN) 28f, **103**
Save (SAVE) 28f, **103**
Scale (FSCAL) **84**
Select Initial BMC Map Entry (DOB BMC) **68**
Select Initial Map Register (DOB BMC) **68**
Set Bit to One (BTO) **54**
Set Bit to Zero (BTZ) **54**
Set S to One (DSSO) **122**
Set S to Zero (DSSZ) **122**
Set T to One (DSTO) **122**
Set T to Zero (DSTZ) **122**

Instructions
Sign Extend and Divide (DIVX) **67**
Signed Divide (DIVS) **66**
Signed Multiply (MULS) **98**
Skip Always (FSA) **83**
Skip if ACS Greater than ACD (SGT) **104**
Skip if ACS Greater than or Equal to ACD (SGE) **104**
Skip on Greater Than or Equal To Zero (FSGE) **84**
Skip on Greater Than Zero (FSGT) **84**
Skip on Less Than or Equal To Zero (FSLE) **84**
Skip on Less Than Zero (FSLT) **84**
Skip on No Error (FSNER) **85**
Skip on No Mantissa Overflow (FSNM) **85**
Skip on No Overflow (FSNO) **85**
Skip on No Overflow and No Zero Divide (FSNOD) **86**
Skip on No Undeflow (FSNU) **86**
Skip on No Undeflow and No Zero Divide (FSUD) **86**
Skip on No Undeflow and Overflow (FSNUO) **86**
Skip on No Zero Divide (FSND) **85**
Skip on Nonzero (FSNE) **85**
Skip on Nonzero Bit (SNB) **85**
Skip on Zero (FSEQ) **84**
Skip on Zero Bit (SZB) **107**
Skip on Zero Bit and Set to One (SZBO) **107**
Specify BMC Map Entry Count (DOC BMC) **69**
Specify BMC Map Table Transfer (DOB BMC) **68**
Specify Initial Address (DOA BMC) **68**
Specify Low-Order Address (DOA BMC) **68**
Specify Operation and High-Order Address (DOB BMC) **68**
Store Accumulator (STA) **105**
Store Byte (STB) **105**
Store Floating-Point Double (FSTD) **87**
Store Floating-Point Single (FSTS) **87**
Store Floating-Point Status (FSST) **86**
Store in Stack (DSTK) **122**
Store Integer (STI) **124**
Store Integer Extended (STIX) **125**
Subtract (SUB) **106**
Subtract Double, FPAC from FPAC (FSD) **84**
Subtract Double, Memory from FPAC (FSMD) **85**
Subtract Immediate (SBI) **104**
Subtract Single, FPAC from FPAC (FSS) **86**
Subtract Single, Memory from FPAC (FSMS) **85**
System Call (SYC) **107**
Translate Page 31 (DOB MAP) **68**
Translate User Single Cycle (NIOP MAP) **100**
Trap Disable (FTD) **87**
Trap Enable (FTE)) **87**

Instructions
Unsigned Divide (DIV) **65**
Unsigned Multiply (MUL) **97**
Vector on Interrupting Device (VCT) 39, **108f**
Integerize (FINT) instruction **79**
Interrupt Acknowledge (INTA) instruction **91**
Interrupt Disable (INTDS) instruction **91**
Interrupt Enable (INTEN) instruction **91**
Interrupt On flag 36f
Interrupt system
instruction list 37
Interrupt On flag 36f
interrupts 37f
instruction interruption 37
level 38
mask 38
servicing 38
vector interrupt processing 39
ION flag. *See* Interrupt On flag.

J

Jump (JMP) instruction **92**
Jump instructions 25
Jump to Subroutine (JSR) instruction 28f, **92**

L

LEF flag. *See* Load Effective Address flag.
Load Accumulator (LDA) instruction **93**
Load Byte (LDB) instruction **93**
Load Effective Address flag 35
Load Effective Address (LEF) instruction **93**
Load Exponent (FEXP) instruction **77**
Load Floating-Point Double (FLDD) instruction **79**
Load Floating-Point Single (FLDS) instruction **80**
Load Floating-Point Status (FLST) instruction **80**
Load Integer (LDI) instruction **123**
Load Integer Extended (LDIX) instruction **123**
Load Map (LMP) instruction **94**
Load Sign (LSN) instruction **124**
Load User/DCH Map Table (LMP) instruction **94**
Load User/DCH Translator Status (DOA MAP) instruction **68**
Locate and Reset Lead Bit (LRB) instruction **94**
Locate Lead Bit (LOB) instruction **94**
Logical Shift (LSH) instruction **94**
Logical
data formats 13f
instruction lists 17f
Lower page zero 6f

M

Management
device 4, 35f
memory 4, 41f
program flow 4, 25f
stack 1
system 4, 45
Mantissa alignment 20
MAP. *See* Memory allocation and protection
Map Single Cycle (NIOP MAP) instruction **100**
Map Supervisor Page 31 (DOB MAP) instruction **68**
Map table entry formats 41f
Mask Out (MSKO) instruction **96**
Memory reference instruction format 5f
Memory
access 1, 5f
allocation and protection 41
fault handling 43f
management 4, 41f
protection 9, 43
reserved locations 6f
system 2
Modify Stack Pointer (MSP) instruction **96**
Move (MOV) instruction **95**
Move Alphabetics (DMVA) instruction **120**
Move Characters (DMVC) instruction **121**
Move Digit with Overpunch (DMVO) instruction **121**
Move Float (DMFV) instruction **121**
Move Floating-Point (FMOV) instruction **81**
Move Numeric with Zero Suppression (DMVS) instruction **122**
Mover Numerics (DMVN) instruction **121**
Move
instruction lists
commercial 117
fixed-point 15, 18
floating-point 21
Multiplication
instruction lists
fixed-point 16
floating-point 22
operation, floating-point 21
Multiply Double, FPAC by FPAC (FMD) instruction **80**
Multiply Double, FPAC by Memory (FMMD) instruction **81**
Multiply Single, FPAC by FPAC (FMS) instruction **81**
Multiply Single, FPAC by Memory (FMMS) instruction **81**

N

Negate (FNEG) instruction **82**
Negate (NEG) instruction **99**
No I/O Transfer (NIO) instruction **100**
Normalization 20 Normalize (FNOM) instruction **82**
No Skip (FNS) instruction **82**

O

Op code field 5
Operand accessing 7f
Operations
floating-point 19f
stack 9f, 12, 28f

P

Page 31 register 42
Page Check (DIC MAP) instruction **64**
Pop Block (POPB) instruction **101**
Pop Floating-Point State (FPOP) instruction **82**
Pop Multiple Accumulators (POP) instruction **100**
Pop PC and Jump (POPJ) instruction **101**
Priority interrupt mask 38
Processor 2
Program counter 4
Program-counter-relative addressing 6
Program flow 4, 25f
Programmed I/O 4
Protection
 fault handling 43f
 indirection 43
 input/output 43
 memory 9, 43, 45
 stack 11f
 validity 43
 write 43
Push Floating-Point State (FPSH) instruction **83**
Push Jump (PSHJ) instruction **101**
Push Multiple Accumulators (PSH) instruction **101**
Push Return Address (PSHR) instruction **102**

R

Read BMC Status (DIB BMC) instruction **64**
Read High Word (FRH) instruction **83**
Read Memory Fault Address (DIA ERCC) instruction **63**
Read Memory Fault Code and Address (DIB ERCC) instruction **64**
Read Switch Register (READS) instruction **102**
Read User/DCH Translator Status (DIA MAP) instruction **63**
Register formats
 accumulator 3
 floating-point status 23
 map table entries 41f
 page 31 42
 program counter 4
 stack parameters 9f
Reserved memory locations 6f
Restore (RSTR) instruction **102**
Return block 11
Return (RTN) instruction 28f, **103**

S

Save (SAVE) instruction 28f, **103**
Scale (FSCAL) instruction **84**
Select Initial BMC Map Entry (DOB BMC) instruction **68**
Select Initial Map Register (DOB BMC) instruction **68**
Set Bit to One (BTO) instruction **54**
Set Bit to Zero (BTZ) instruction **55**
Set S to One (DSSO) instruction **122**
Set S to Zero (DSSZ) instruction **122**
Set T to One (DSTO) instruction **122**
Set T to Zero (DSTZ) instruction **122**
Shift
 instruction lists 17f
 operations 14f
Sign Extend and Divide (DIVX) instruction **67**
Sign instructions 117
Signed Divide (DIVS) instruction **66**
Signed Multiply (MULS) instruction **98**
Single-precision data formats
 fixed-point 13
 floating-point 19
Skip Always (FSA) instruction **83**
Skip if ACS Greater than ACD (SGT) instruction **104**
Skip if ACS Greater than or Equal to ACD (SGE) instruction **104**
Skip on Greater Than or Equal To Zero (FSGE) instruction **84**
Skip on Greater Than Zero (FSGT) instruction **84**
Skip on Less Than or Equal To Zero (FSLE) instruction **84**
Skip on Less Than Zero (FSLT) instruction **84**
Skip on No Error (FSNER) instruction **85**
Skip on No Mantissa Overflow (FSNM) instruction **85**
Skip on No Overflow (FSNO) instruction **85**
Skip on No Overflow and No Zero Divide (FSNOD) instruction **86**
Skip on No Undeflow (FSNU) instruction **86**
Skip on No Undeflow and No Zero Divide (FSUD) instruction **86**
Skip on No Undeflow and Overflow (FSNUO) instruction **86**
Skip on No Zero Divide (FSND) instruction **85**
Skip on Nonzero (FSNE) instruction **85**
Skip on Nonzero Bit (SNB) instruction **105**
Skip on Zero (FSEQ) instruction **84**
Skip on Zero Bit (SZB) instruction **107**
Skip on Zero Bit and Set to One (SZBO) instruction **107**
Skip
 instruction lists 17f, 23
 legal and illegal instruction sequences 26
 operations 15
Specify BMC Map Entry Count (DOC BMC) instruction **69**
Specify BMC Map Table Transfer (DOB BMC) instruction **68**
Specify Initial Address (DOA BMC) instruction **68**
Specify Low-Order Address (DOA BMC) instruction **68**
Specify Operation and High-Order Address (DOB BMC) instruction **68**
Stack faults. *See* Stack protection.
Stack
 fault handling 32f
 initialization 11f

instruction lists
return block 11
word access 10
parameter 10
management 1
operations 9f, 12, 28f
parameters 9f
frame pointer 10
base 9
limit 9
pointer 10
protection 11f
return block 11
Status
address translator 44
floating-point 23
system 45
Store Accumulator (STA) instruction **105**
Store Byte (STB) instruction **105**
Store Floating-Point Double (FSTD) instruction **87**
Store Floating-Point Single (FSTS) instruction **87**
Store Floating-Point Status (FSST) instruction **86**
Store in Stack (DSTK) instruction **122**
Store Integer (STI) instruction **124**
Store Integer Extended (STIX) instruction **125**
Subroutines
examples 30f
instruction list 27f
instruction sequence and types 27f
returns 27f
Subtract (SUB) instruction **106**
Subtract Double, FPAC from FPAC (FSD) instruction **84**
Subtract Double, Memory from FPAC (FSMD) instruction **85**
Subtract Immediate (SBI) instruction **104**
Subtract Single, FPAC from FPAC (FSS) instruction **86**
Subtract Single, Memory from FPAC (FSMS) instruction **85**
Subtraction
instruction lists 16, 21
operations, floating-point 21
System Call (SYC) instruction **107**
System
block diagram 2
functions 45
input/output 2
memory 2
overview 1f
status 45

T

Translate Page 31 (DOB MAP) instruction **68**
Translate User Single Cycle (NIOP MAP) instruction **100**
Trap Disable (FTD) instruction **87**
Trap Enable (FTE) instruction **87**
Truncation 20

U

Unsigned Divide (DIV) instruction **65**
Unsigned Multiply (MUL) instruction **97**

V

Validity protection 43
Vector interrupt processing 39
Vector on Interrupting Device Code (VCT) instruction 39, **108f**

W

Word access 7
Write protection 43

DG OFFICES

NORTH AMERICAN OFFICES

Alabama: Birmingham
Arizona: Phoenix, Tucson
Arkansas: Little Rock
California: Anaheim, El Segundo, Fresno, Los Angeles, Oakland, Palo Alto, Riverside, Sacramento, San Diego, San Francisco, Santa Barbara, Sunnyvale, Van Nuys
Colorado: Colorado Springs, Denver
Connecticut: North Branford, Norwalk
Florida: Ft. Lauderdale, Orlando, Tampa
Georgia: Norcross
Idaho: Boise
Iowa: Bettendorf, Des Moines
Illinois: Arlington Heights, Champaign, Chicago, Peoria, Rockford
Indiana: Indianapolis
Kentucky: Louisville
Louisiana: Baton Rouge, Metairie
Maine: Portland, Westbrook
Maryland: Baltimore
Massachusetts: Cambridge, Framingham, Southboro, Waltham, Wellesley, Westboro, West Springfield, Worcester
Michigan: Grand Rapids, Southfield
Minnesota: Richfield
Missouri: Creve Coeur, Kansas City
Mississippi: Jackson
Montana: Billings
Nebraska: Omaha
Nevada: Reno
New Hampshire: Bedford, Portsmouth
New Jersey: Cherry Hill, Somerset, Wayne
New Mexico: Albuquerque
New York: Buffalo, Lake Success, Latham, Liverpool, Melville, New York City, Rochester, White Plains
North Carolina: Charlotte, Greensboro, Greenville, Raleigh, Research Triangle Park
Ohio: Brooklyn Heights, Cincinnati, Columbus, Dayton
Oklahoma: Oklahoma City, Tulsa
Oregon: Lake Oswego
Pennsylvania: Blue Bell, Lancaster, Philadelphia, Pittsburgh
Rhode Island: Providence
South Carolina: Columbia
Tennessee: Knoxville, Memphis, Nashville
Texas: Austin, Dallas, El Paso, Ft. Worth, Houston, San Antonio
Utah: Salt Lake City
Virginia: McLean, Norfolk, Richmond, Salem
Washington: Bellevue, Richland, Spokane
West Virginia: Charleston
Wisconsin: Brookfield, Grand Chute, Madison

DG-04976

INTERNATIONAL OFFICES

Argentina: Buenos Aires
Australia: Adelaide, Brisbane, Hobart, Melbourne, Newcastle, Perth, Sydney
Austria: Vienna
Belgium: Brussels
Bolivia: La Paz
Brazil: Sao Paulo
Canada: Calgary, Edmonton, Montreal, Ottawa, Quebec, Toronto, Vancouver, Winnipeg
Chile: Santiago
Columbia: Bogata
Costa Rica: San Jose
Denmark: Copenhagen
Ecuador: Quito
Egypt: Cairo
Finland: Helsinki
France: Le Plessis-Robinson, Lille, Lyon, Nantes, Paris, Saint Denis, Strasbourg
Guatemala: Guatemala City
Hong Kong
India: Bombay
Indonesia: Jakarta, Pusat
Ireland: Dublin
Israel: Tel Aviv
Italy: Bologna, Florence, Milan, Padua, Rome, Tourin
Japan: Fukuoka, Hiroshima, Nagoya, Osaka, Tokyo, Tsukuba
Jordan: Amman
Korea: Seoul
Kuwait: Kuwait
Lebanon: Beirut
Malaysia: Kuala Lumpur
Mexico: Mexico City, Monterrey
Morocco: Casablanca
The Netherlands: Amsterdam, Rijswijk
New Zealand: Auckland, Wellington
Nicaragua: Managua
Nigeria: Ibadan, Lagos
Norway: Oslo
Paraguay: Asuncion
Peru: Lima
Philippine Islands: Manila
Portugal: Lisbon
Puerto Rico: Hato Rey
Saudi Arabia: Jeddah, Riyadh
Singapore
South Africa: Cape Town, Durban, Johannesburg, Pretoria
Spain: Barcelona, Bibao, Madrid
Sweden: Gothenburg, Malmo, Stockholm
Switzerland: Lausanne, Zurich
Taiwan: Taipei
Thailand: Bangkok
Turkey: Ankara
United Kingdom: Birmingham, Bristol, Glasgow, Hounslow, London, Manchester
Uruguay: Montevideo
USSR: Espoo
Venezuela: Maracaibo
West Germany: Dusseldorf, Frankfurt, Hamburg, Hannover, Munich, Nuremburg, Stuttgart

Data General

Ordering Technical Publications

TIPS is the Technical Information and Publications Service—a new support system for DGC customers that makes ordering technical manuals simple and fast. Simple, because TIPS is a central supplier of literature about DGC products. And fast, because TIPS specializes in handling publications.

TIPS was designed by DG's Educational Services people to follow through on your order as soon as it's received. To offer discounts on bulk orders. To let you choose the method of shipment you prefer. And to deliver within a schedule you can live with.

How to Get in Touch with TIPS

Contact your local DGC education center for brochures, prices, and order forms. Or get in touch with a TIPS administrator directly by calling (617) 366-8911, extension 4086, or writing to

Data General Corporation
Attn: Educational Services, TIPS Administrator
MS F019
4400 Computer Drive
Westborough, MA 01580

TIPS. For the technical manuals you need, when you need them.

DGC Education Centers

Boston Education Center
Route 9
Southboro, Massachusetts 01772
(617) 485-7270

Washington, D.C. Education Center
7927 Jones Branch Drive, Suite 200
McLean, Virginia 22102
(703) 827-9666

Atlanta Education Center
6855 Jimmy Carter Boulevard, Suite 1790
Norcross, Georgia 30071
(404) 448-9224

Los Angeles Education Center
5250 West Century Boulevard
Los Angeles, California 90045
(213) 670-4011

Chicago Education Center
703 West Algonquin Road
Arlington Heights, Illinois 60005
(312) 364-3045

134-755-01

moisten & seal

CUSTOMER DOCUMENTATION COMMENT FORM

Your Name ______________________ Your Title ______________

Company ______________________________________

Street ______________________________________

City ______________________ State __________ Zip ______

We wrote this book for you, and we made certain assumptions about who you are and how you would use it. Your comments will help us correct our assumptions and improve the manual. Please take a few minutes to respond. Thank you.

Manual Title ______________________ Manual No. ______________

Who are you?
☐EDP/MIS Manager
☐Senior Systems Analyst
☐Engineer
☐Analyst/Programmer
☐Operator
☐End User
☐Other ______________

How do you use this manual? *(List in order: 1 = Primary Use)*

___ Introduction to the product
___ Reference
___ Tutorial Text
___ Operating Guide
___ Other ______________

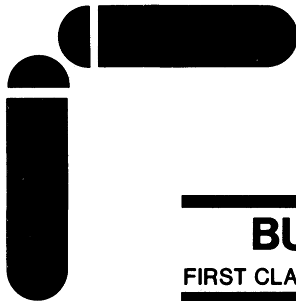
fold

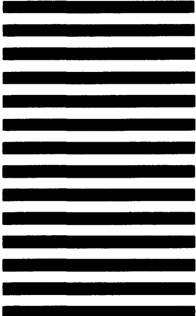
About the manual:

	Yes	No
Is it easy to read?	☐	☐
Is it easy to understand?	☐	☐
Are the topics logically organized?	☐	☐
Is the technical information accurate?	☐	☐
Can you easily find what you want?	☐	☐
Does it tell you everything you need to know?	☐	☐
Do the illustrations help you?	☐	☐

If you wish to order manuals, use the enclosed TIPS Order Form (USA only).

Comments:





BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 26 SOUTHBORO, MA 01772

Postage will be paid by addressee

Customer Documentation
MS E-219
4400 Computer Drive
Westboro, MA 01581-9973

134-755-01

Data General Corporation, Westboro, MA 01580

014-000688-02