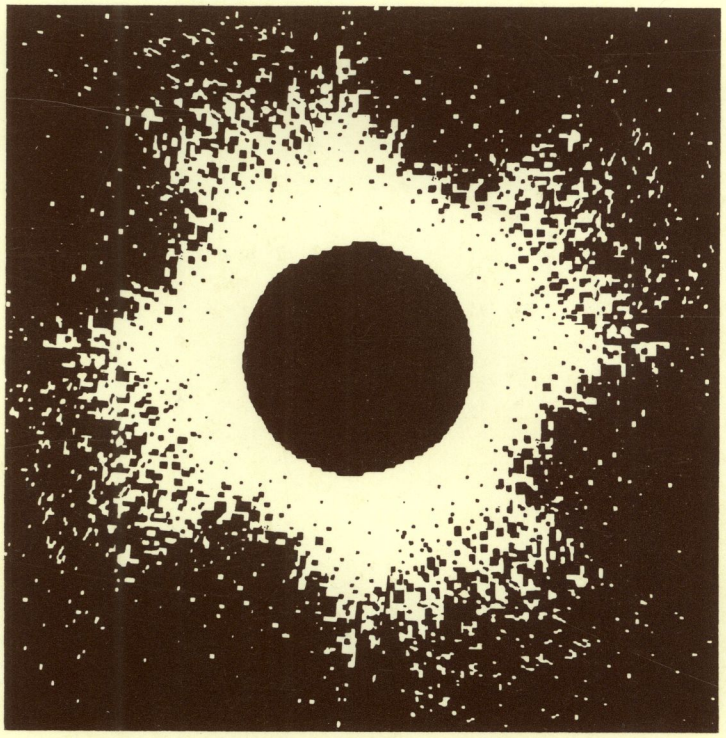
Data General

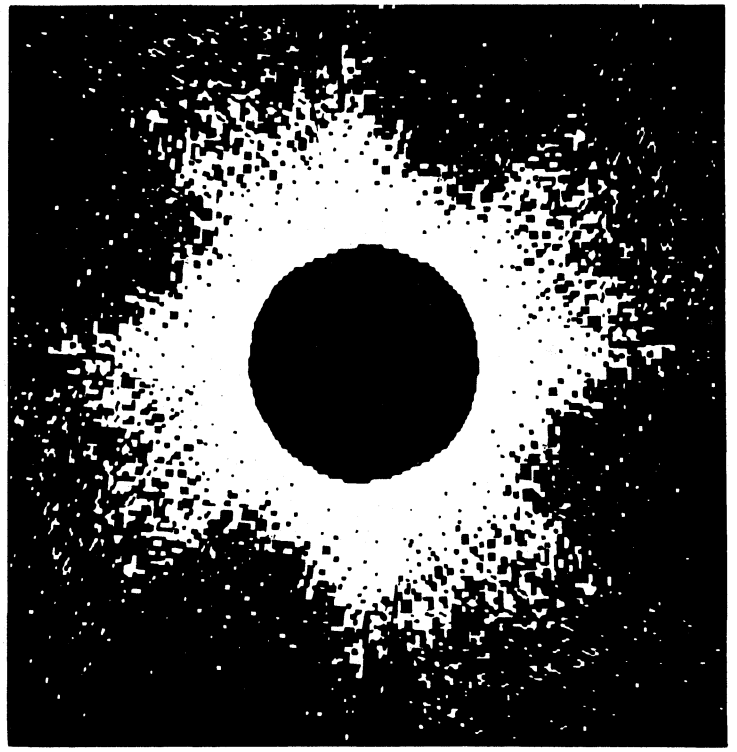


ECLIPSE® S/140 Programmer's Reference

Warning: For devices compliant with FCC Rules

This equipment generates, uses, and can radiate radio frequency energy and if not installed and used in accordance with the instruction manual, may cause interference to radio communications. It has been tested and found to comply with the limits for Class A computing devices pursuant to Subpart J of Part 15 of FCC Rules, which are designed to provide reasonable protection against such interference when operated in a commercial environment. Operation of this equipment in a residential area is likely to cause interference, in which case the user, at his own expense, will be required to take whatever measures may be required to correct the interference.

Data General



ECLIPSE® S/140 Programmer's Reference

NOTICE

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

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

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

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

Ordering No. 014-000642

Printed in the United States of America
Rev. 04, September 1986

CONTENTS

Chapter 1
Introduction to the ECLIPSE S/140

The Data General ECLIPSE S/140 is a scientific computer combining advanced architecture and high reliability. Four main components make up the ECLIPSE S/140, providing processing power and throughput capability. The components are:

- Main Storage system
- I/O Management system
- Main Processor
- Packaging

In addition, a generous selection of software supports the S/140 system, and the S/140 is compatible with DGC's NOVA computers.

Main Storage

Maximum memory capacity of the ECLIPSE S/140 is 2 Mbytes (up to eight boards) in the form of semiconductor RAM. Each memory board contains four modules that support cycle times as low as 100 nanoseconds for a read operation and 200 nanoseconds for a write operation.

The ECLIPSE S/140 Memory Management and Protection Unit (MMPU) provides and protects individual user space within memory on a 2 Kbyte page basis. Protection modes include address validity, indirect, write, and I/O protections.

The Error Checking and Correction (ERCC) facility detects and corrects all single bit errors that occur on a memory board. The ERCC detects and reports errors by maintaining address and fault codes and requesting processor interrupts when errors occur. Memory cycle times are left unchanged if no error is detected. If an error is found, cycle time is increased by 200 nanoseconds.

I/O Management

The S/140 is a powerful and highly reliable scientific computer. The advanced architectural features of this system provide configurational flexibility to match various computing needs. ECLIPSE S/140 supports an optional Burst Multiplexor Channel for high-speed data transfers.

The standard NOVA/ECLIPSE data channel provides I/O communication for both medium-speed and high-speed devices such as cartridge discs, magnetic tape, data channel line printers, and synchronous communications. Maximum data channel transfer rates are 2.0 Mbytes per second fast input, 1.4 Mbytes per second fast output.

Programmed I/O, with priority interrupt handling and vectoring capability for automatic dispatch to the correct interrupt handler, provides I/O communication for low-speed devices such as CRT terminals, paper tape punches, and card readers.

Main Processor

The ECLIPSE S/140 main processor executes the standard ECLIPSE instruction set. Integer multiply/divide functions are implemented in firmware.

The main processor also executes the optional ECLIPSE floating point instruction set, using either Floating Point microcode, or the high-speed hardware Floating Point Processor.

The Character Instruction Set (CIS) simplifies the handling of strings of characters or bytes. It is especially useful in communications applications where long strings of bytes must be moved, compared, or checked against a reference.

On power up, the processor executes a self-test; if this test is successfully completed, the CPU enters virtual console mode. The virtual console replaces all but three switches on the front panel of the ECLIPSE S/140. Only the Power On/Off, Lock, and Reset/Program Load function switches remain.

Packaging

The ECLIPSE S/140 is packaged in an easily accessed chassis that holds up to sixteen 15" x 15" printed circuit boards. The power supply consists of a 100-amp VNR unit and a slide-in power supply board. Battery back-up is standard.

Software Support

Two proven operating systems and many advanced utilities and high level languages are available for the ECLIPSE S/140.

The Real-Time Disc Operating System (RDOS) supports real-time and batch operations, along with independent foreground/background processing. RDOS can manage up to 2 Mbytes of main memory in the ECLIPSE S/140.

The Advanced Operating System (AOS) uses adaptive resource management for efficient operation in multiuser environments. It can manage up to 2 Mbytes of main memory in the ECLIPSE S/140 and supports concurrent batch, timesharing, and real-time operations.

Many higher-level languages are also available, including FORTRAN IV, FORTRAN 5, Extended BASIC, PL/1, DG/L (an ALGOL-derivative, structured programming language), and Macroassembler.

ECLIPSE/NOVA Line Compatibility

The ECLIPSE S/140 is compatible with Data General's NOVA line computers. You may execute any NOVA-based program on an ECLIPSE series computer, if several criteria are met. First, the program must not be dependent on instruction execution times or I/O transfer times, because ECLIPSE times may be faster. Second, the ECLIPSE system must contain at least as much memory and an equal number of I/O devices as your NOVA system. Third, it is important that your NOVA-based program *does not* use any of the following:

- ACL instructions specifying both the *no load* and the *never skip* options.
- The NOVA Memory Management and Protection Unit.
- The data channel increment or add-to-memory features.
- The instructions **PUSH, POPA, SAV, MTSP, MTFP, MFSP, MFFP, LDB, STB, RET.**

NOVA and ECLIPSE multiply/divide operations function the same way. However, each uses a different operation code.

NOVA and ECLIPSE floating point instructions are similar, but do not function the same way. Remember to check the instructions formats. Floating point data formats are the same for both machines.

Chapter 2
Standard Features

In this chapter we discuss the standard features of the ECLIPSE S/140, and the assembly language instructions controlling these facilities. In the following chapter we describe optional facilities and their instructions. Chapters 4 and 5 contain complete instruction descriptions for standard and I/O instructions, respectively.

After explaining the standard addressing conventions of the ECLIPSE S/140, we discuss the functions of

Bit manipulation
Byte manipulation
Number manipulation
Logical manipulation
ALC manipulation
The stack
Reserved storage locations
Program execution
Extended operations
I/O operation, interrupts, and vectoring
I/O devices
Power fail/auto-restart
Error checking and correction
The virtual console and
Memory management and protection.

Addressing Conventions

In this section we describe the various ways the ECLIPSE S/140 addresses locations in memory. We also define terms and concepts useful for understanding the addressing process in the ECLIPSE S/140.

Each addressed location in main memory consists of a 16-bit word. The first word in memory has the address 0, the next has the address 1, the next 2, and so forth.

The maximum amount of *logical* address space available to the programmer is 32,768 words. The *physical* address space, the amount of memory in the system, may be much larger. In the logical address space, the next sequential memory location after 77777_8 is location 0.

The MMPU controls the relationship between a logical address space and the physical address space. When the MMPU is enabled, it intercepts each memory reference and translates the 15-bit logical address into a 20-bit physical address. Unless the MMPU itself is being programmed, the translation process is invisible to the programmer.

There are three modes of addressing: Absolute, P.C. Relative, and Accumulator Relative. You may use direct or indirect addressing in any of these modes. The following definitions are useful in understanding ECLIPSE S/140 addressing conventions.

Direct and Indirect Addressing — Direct addressing uses the intermediate address (the first address found) without modification. The intermediate address thus becomes the effective address.

Indirect addressing uses the intermediate address as a pointer to the next address. If bit 0 of that next address is 1, the address is used as a *pointer* to another address. A series of indirect addresses is called an *indirection chain*. The chain continues until an address is found with bit 0 equal to 0. Indirect protection is available to limit indirection levels to fifteen.

Indirect Bit — This is a bit in the instruction or address that is checked after each address calculation. If the indirect bit is 0, the effective address has been located. If it is 1, the word contains another indirect address.

Index Bits — These are bits in the instruction that control which of the three addressing modes the instruction uses. The correlation between index bits and addressing mode is shown in Table 1.1.

Index Bits	Mode
00	Absolute
01	PC relative
10	AC2 relative
11	AC3 relative

Table 1.1 Address mode selection

Displacement Bits — These are bits in the instruction that specify an address in memory. This address, added to an address specified by the mode, results in the effective address.

When the index bits are 00, the displacement is considered an unsigned integer. When the index bits are 01, 10, or 11, the displacement is considered a signed integer. Table 2.2 shows the range of the displacement field under various conditions.

Index Bits	Range of Displacement Field	
	Short Class	Extended Class
00	0 to 377_8 or 0 to 255_{10}	0 to 77777_8 or 0 to $32,767_{10}$
01	-200_8 to 177_8 or	-40000_8 to 37777_8 or
10	-128 to $+127_{10}$	-16,384 to $+16,383_{10}$
11		

Table 2.2 Ranges of displacement

Effective Address Calculation — This process converts the index, indirect, and displacement bits into an address to be used by the instruction.

Intermediate Address — This address is obtained by the effective address calculation before the indirect bit is checked.

Page Zero — The locations 0-377_8 in memory comprise page zero.

Addressing Modes

As mentioned earlier, three modes of addressing can be done in the ECLIPSE S/140. They are:

- Absolute addressing
- P.C. (program counter) relative addressing
- Accumulator relative addressing

Figure 2.1 illustrates the three addressing modes.

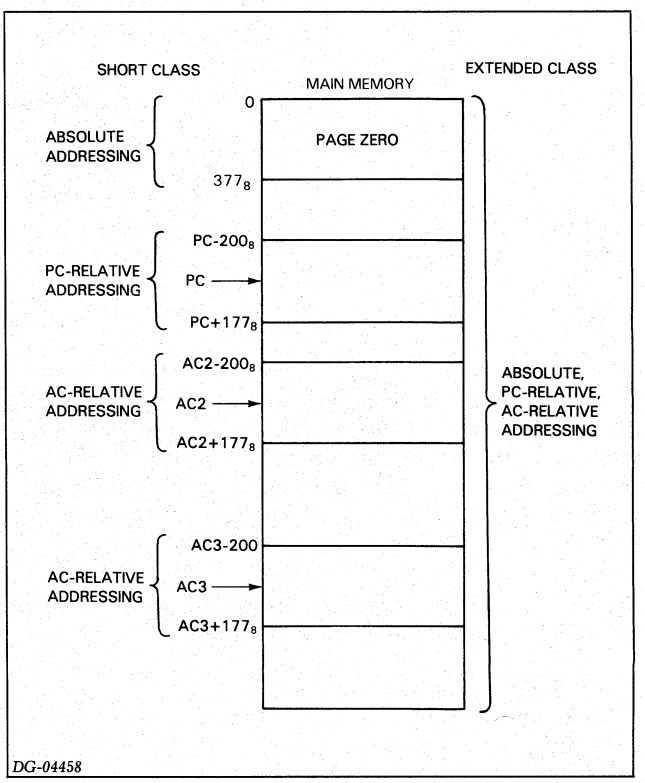


Figure 2.1 Addressing modes

You can use direct or indirect addressing with each of these modes. With the right combination, any address in your logical address space is accessible.

Absolute Addressing Mode — In this mode, the intermediate address is set equal to the unmodified displacement. As a result, short class instructions specify locations in the range 0-377_8 in the absolute mode, because short class instructions are restricted to 8 bits in the displacement. Extended class instructions can reference any logical memory address using the absolute addressing mode.

Page zero thus is very important because any memory-reference instruction can address this area. You can use it as a common storage area for items that you frequently reference throughout a program. Note, however, that DGC reserves some of these locations for special purposes.

P.C. Relative Addressing Mode — The intermediate address is found by adding the displacement to the contents of the program counter.

Accumulator Relative Addressing Mode — The intermediate address is found by adding the displacement to the contents of bits 1-15 of the accumulator indicated by the index bits (you may use either AC2 or AC3).

Auto-incrementing and Auto-decrementing

During indirect addressing, certain reserved locations within the area of $0\text{-}377_8$ (page zero) automatically increment or decrement their contents. The process is also called auto-indexing.

Auto-incrementing takes place if the intermediate address of a short class instruction falls into the range $20\text{-}27_8$, and the indirect bit is 1. The contents of the addressed location are incremented by 1, and the addressing chain continues, using the *incremented* value of the addressed location.

Auto-decrementing takes place if the intermediate address of a short class instruction falls into the range $30\text{-}37_8$, and the indirect bit is 1. The contents of the addressed location are decremented by 1, and the addressing chain continues, using the *decremented* value of the addressed location.

> **NOTE:** *The state of bit 0 before the increment or decrement determines whether the indirection chain is continued. Assume, for example, that an auto-increment location contains 177777_8 (all bits = 1, including bit 0), and the location is referenced as part of an indirection chain. After incrementing, the location contains all zeros. Because bit 0 was 1 before the increment, 000000 is treated as an intermediate address and the indirection chain continues.*

You will find a flow diagram of the addressing process in Appendix A.

Bit Manipulation

We use a 32-bit (2-word) *bit pointer* to address individual bits in memory. Bit 0 of the bit pointer is the indirect bit. If this bit is 1, the indirection chain (using bits 1-15 for the address each time) will be followed until a word is found with bit 0 set to 0. Bits 1-15 of this word become bits 1-15 of the bit pointer, and bits 0-15 of the next word become bits 16-31 of the bit pointer.

To determine the location of a desired bit, the address formed by the unsigned number contained in bits 1-15 of the bit pointer (the *base* address) is added to the number formed by the 12-bit unsigned number contained in bits 16-27 (the *offset*). The resulting address points to the word containing the desired bit. Bits 28-31 of the bit pointer contain a 4-bit unsigned number, which is the number of the desired bit in the addressed word.

Figure 2.2 diagrams the bit addressing process.

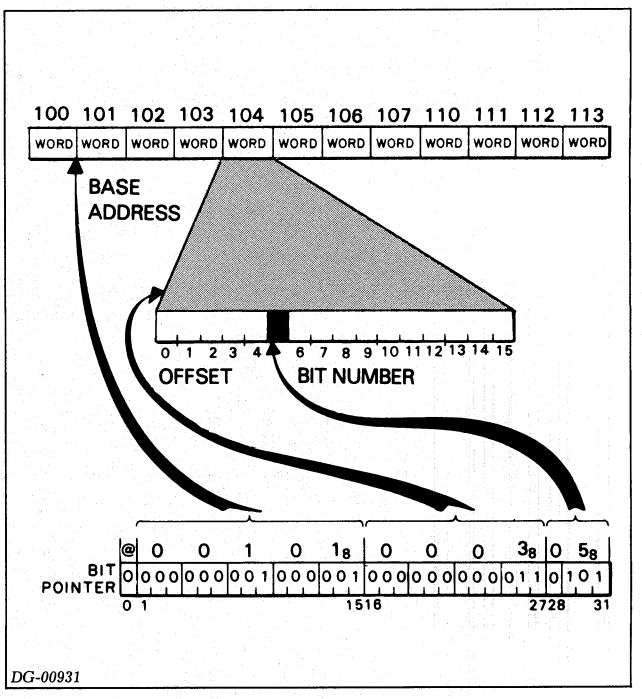


Figure 2.2 Bit addressing process

Bit Instructions

ECLIPSE S/140 instructions that manipulate bits do the following:

- Locate a bit in memory and set it to 0 or 1
- Test a bit, skipping the next word if the specified condition is true
- Add a number to the contents of one accumulator, based on the number of ones or high-order zeros found in the other accumulator

Some bit instructions use a bit pointer to locate a bit in memory. Others only affect bits within specified accumulators. In addition, a character instruction (**CMT**), uses a bit pointer, bit addressing and a bit table. Table 2.3 lists the bit instructions.

Mnem	Instructions	Action
BTO	Set Bit To One	Sets the bit addressed by the bit pointer to 1.
BTZ	Set Bit To Zero	Sets the bit addressed by the bit pointer to 0.
COB	Count Bits	Counts the number of ones in one accumulator and adds that number to the second accumulator.
LOB	Locate Lead Bit	Counts the number of high-order zeros in one accumulator and adds that number to the second accumulator.
LRB	Locate And Reset Lead Bit	Performs a *Locate Lead Bit* instruction and sets the lead bit to 0.
SNB	Skip On Non-Zero Bit	Skips the next sequential word if the bit addressed by the bit pointer is 1.
SZB	Skip On Zero Bit	Skips the next sequential word if the bit addressed by the bit pointer is 0.
SZBO	Skip On Zero Bit And Set To One	Sets the bit addressed by the bit pointer to 1 and skips the next sequential word if the bit was originally 0.

Table 2.3 Bit instructions

Byte Manipulation

A byte in memory is selected by a sixteen bit *byte pointer*. Bits 0-14 of the byte pointer contain the memory address of a two byte word. Bit 15, the *byte indicator*, indicates which byte of the addressed location will be used. If bit 15 is 0, the high-order byte (bits 0-7) will be used. If bit 15 is 1, the low-order byte (bits 8-15) will be used. Figure 2.3 shows the format of a byte pointer.

Byte Instructions

The byte instructions are shown in Table 2.4. Note that when an instruction moves a byte to an accumulator it also clears the high-order half of the destination accumulator. When an instruction moves a byte from an accumulator to memory, it leaves unchanged the other byte contained in that word of memory.

The two extended instructions, (**ELDB** and **ESTB,**) reference bytes with a byte pointer contained in the instruction coding. The two short class instructions (**LDB** and **STB**) use an accumulator to hold the byte pointer. The character instructions also use byte addressing.

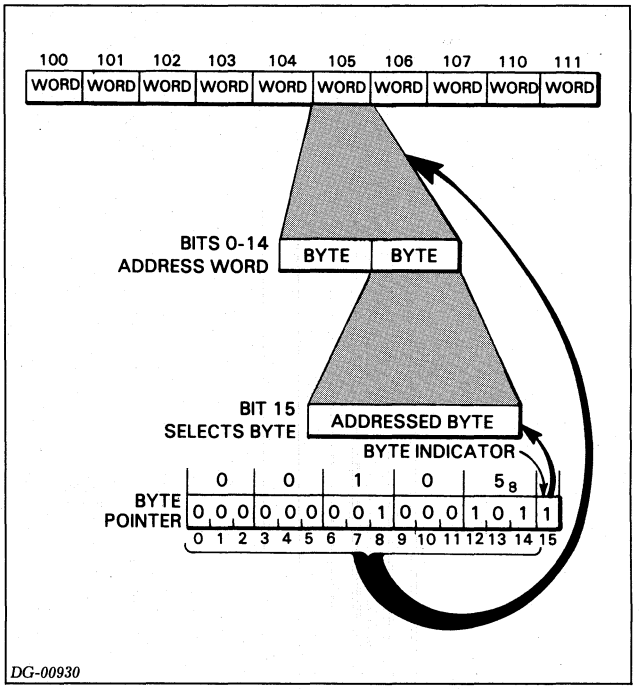


Figure 2.3 Byte pointer format

Mnem	Instructions	Action
LDB ELDB	Load Byte	Places a byte of information into an accumulator.
STB ESTB	Store Byte	Stores the low order byte of an accumulator into a byte of memory.

Table 2.4 Byte instructions

Number Manipulation

We represent a signed integer by a two's-complement number in one or more 16-bit words. The sign of the number is positive if bit 0 of the first word is 0 and negative if that bit is 1.

We represent an unsigned integer by using all the bits of one or more 16-bit words to represent the magnitude. Figure 2.4 illustrates integer format.

Single precision integers are one word (16 bits) long, and multiple precision integers are two or more words long. As an example, Table 2.5 shows the range of single and double precision numbers represented by this format:

In addition, there is a value called *carry*. A change in the value of carry indicates an overflow during fixed point arithmetic operations.

Fixed Point Arithmetic Instructions

There are twenty-six ECLIPSE S/140 instructions which perform fixed point arithmetic. These instructions:

- Perform binary arithmetic on operands in accumulators.
- Load data from memory to an accumulator.
- Store data from an accumulator into memory.

All of the fixed point arithmetic instructions are shown in Table 2.6. Some of the instructions appear in both a short form and a long or extended form. (The prefix "E" indicates an extended instruction form.)

Short form instructions (sixteen bits) directly specify either a memory address from 0 to 377_8, or a small area in memory surrounding the present value of the program counter or an accumulator. Long form instructions, thirty-two bits long, directly specify any address from 0 to 77777_8.

ADI and **ADDI** are short and long forms of the same instruction. The short form adds a 2-bit immediate in the range of 1-4, while the long form adds a 16-bit immediate in the range of -32,768 to +32,767.

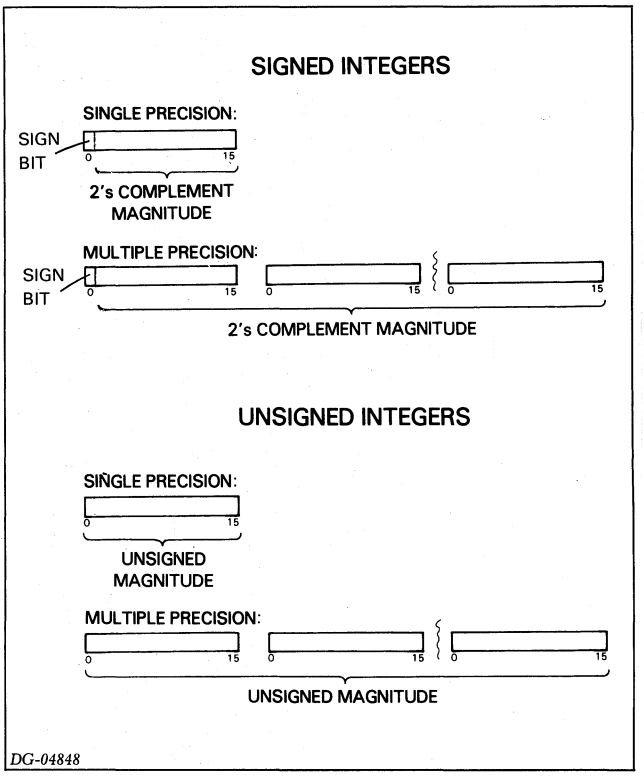


Figure 2.4 Integer format

	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 2.5 Integer ranges

Mnem	Instructions	Action
ADC	Add Complement	Adds the one's complement of the contents of one accumulator to the contents of another accumulator.
ADD	Add	Adds contents of two accumulators.
ADDI	Extended Add Immediate	Adds a signed integer in the range -32,768 to +32,767 to the contents of an accumulator.
ADI	Add Immediate	Adds an unsigned integer in the range 1-4 to the contents of an accumulator.
DIV	Unsigned Divide	Divides the unsigned 32-bit integer in two accumulators by the unsigned contents of a third accumulator.
DIVS	Signed Divide	Divides the signed 32-bit integer in two accumulators by the signed contents of a third accumulator.
DIVX	Sign Extend And Divide	Extends the sign of one accumulator into a second accumulator and performs a *Signed Divide* on the result.
DSZ EDSZ	Decrement And Skip If Zero	Decrements the addressed word, then skips if the decremented value is zero.
HLV	Halve	Divides the unsigned contents of an accumulator by 2.
INC	Increment	Increments the contents of an accumulator.
ISZ EISZ	Increment And Skip If Zero	Increments the addressed word, then skips if the incremented value is zero.
LDA ELDA	Load Accumulator	Loads data from memory to an accumulator.
LEF ELEF	Load Effective Address	Places an effective address in an accumulator.
MOV	Move	Moves the contents of an accumulator through the Arithmetic Logic Unit (ALU).
MUL	Unsigned Multiply	Multiplies the unsigned contents of two accumulators and adds the results to the unsigned contents of a third accumulator.
MULS	Signed Multiply	Multiplies the signed contents of two accumulators and adds the results to the signed contents of a third accumulator.
NEG	Negate	Forms the two's complement of the contents of an accumulator.
SBI	Subtract Immediate	Subtracts an unsigned integer in the range 1-4 from the contents of an accumulator.
STA, ESTA	Store Accumulator	Stores data in memory from an accumulator.
SUB	Subtract	Subtracts contents of one accumulator from another.
XCH	Exchange Accumulators	Exchanges the contents of two accumulators.

Table 2.6 Fixed point arithmetic instructions

Decimal Arithmetic Instructions

Unsigned decimal numbers are handled one decimal digit at a time. Each decimal digit is represented by bits 12-15 of a 16-bit word. Only the values $0\text{-}9_{16}$ are used; carry is used for a decimal carry or borrow.

Two instructions in the ECLIPSE S/140 operate on decimal data. They are shown below, in Table 2.7.

Mnem	Instructions	Action
DAD	Decimal Add	Adds together the decimal digits found in bits 12-15 of two accumulators.
DSB	Decimal Subtract	Subtracts the decimal digit in bits 12-15 of one accumulator from the decimal digit in bits 12-15 of another accumulator.

Table 2.7 Decimal arithmetic instructions

Logical Manipulation

We represent logical entities as individual bits in a 16-bit word. Each bit is treated as a separate binary value. When an instruction operates on two words, only corresponding bits of each word interact. The following are examples of logical operations:

- Forming the logical **AND** of two words.
- Forming the logical complement of a word.
- Shifting the contents of a word left or right.

Logical Operations Instructions

All of the logical operations instructions are shown in Table 2.8.

The *Load Effective Address* and *Extended Load Effective Address* instructions are short and long forms of the same instruction. The sixteen bit short form directly specifies either a memory address from 0 to 255 or a small area in memory surrounding the present value of the program counter or an accumulator. The thirty-two bit long form directly specifies any address from 0 to 77777_8.

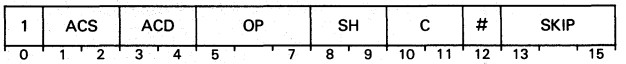
Mnem	Instructions	Action
ANC	AND With Complemented Source	Forms the logical AND of the contents of one accumulator and the logical complement of the contents of another accumulator.
AND	AND	Forms the logical AND of the contents of two accumulators.
ANDI	AND Immediate	Forms the logical AND of a 16-bit number contained in the instruction and the contents of an accumulator.
COM	Complement	Forms the logical complement of the contents of an accumulator.
DHXL	Double Hex Shift Left	Shifts the 32-bit contents of two accumulators left 1 to 4 hex digits depending on the value of a 2-bit number contained in the instruction.
DHXR	Double Hex Shift Right	Shifts the 32-bit contents of two accumulators right 1 to 4 hex digits depending on the value of a 2-bit number contained in the instruction.
DLSH	Double Logical Shift	Shifts the 32-bit contents of two accumulators left or right depending on the contents of a third accumulator.
HXL	Hex Shift Left	Shifts the contents of an accumulator left 1 to 4 hex digits depending on the value of a 2-bit number contained in the instruction.
HXR	Hex Shift Right	Shifts the contents of an accumulator right 1 to 4 hex digits depending on the value of a 2-bit number contained in the instruction.
IOR	Inclusive OR	Forms the logical inclusive OR of the contents of two accumulators.
IORI	Inclusive OR Immediate	Forms the logical inclusive OR of a 16-bit number contained in the instruction and the contents of an accumulator.
LEF ELEF	Load Effective Address	Places an effective address in an accumulator.
LSH	Logical Shift	Shifts the contents of an accumulator left or right depending on the contents of another accumulator.
XOR	Exclusive OR	Forms the logical exclusive OR of the contents of two accumulators.
XORI	Exclusive OR Immediate	Forms the logical exclusive OR of a 16-bit number contained in the instruction and the contents of an accumulator.

Table 2.8 Logical operations instructions

ALC Manipulation

Each of the eight Arithmetic/Logic Class (ALC) instructions performs a specific function upon the contents of one or two accumulators and carry. The eight functions are *Add, Subtract, Negate, Add Complement, Move, AND, Complement,* and *Increment.* The instructions are identified by the mnemonics of the eight functions, which are **ADD, SUB, NEG, ADC, MOV, AND, COM**, and **INC**.

In addition to the specific functions performed by an individual instruction, there is a group of general functions all ALC instructions can perform. These general functions include shift operations, which rotate the data left or right, or swap the bytes. Also included are various tests that can be performed on the data. With each test the instructions can check the data for some condition and skip or not skip the next sequential word, depending on the outcome of the test. Finally, the instructions can load or not load the results of the specific and general functions into the destination accumulator and carry. The diagram below shows the format of the ALC instructions.



ALC Instructions

The ALC instructions are listed in Table 2.9. These instructions use an Arithmetic Logic Unit (ALU) to process data. The logical organization of the ALU is illustrated in Figure 2.5.

Mnem	Instructions	Action
ADC	Add Complement	Adds an unsigned integer to the logical complement of another unsigned number.
ADD	Add	Adds contents of one accumulator to the contents of another.
AND	AND	Forms the logical AND of the contents of two accumulators.
COM	Complement	Forms the logical complement of the contents of an accumulator.
INC	Increment	Increments the contents of an accumulator.
MOV	Move	Moves the contents of an accumulator through the ALU.
NEG	Negate	Forms the two's complement of the contents of an accumulator.
SUB	Subtract	Subtracts contents of one accumulator from the contents of another.

Table 2.9 Arithmetic/logic class instructions

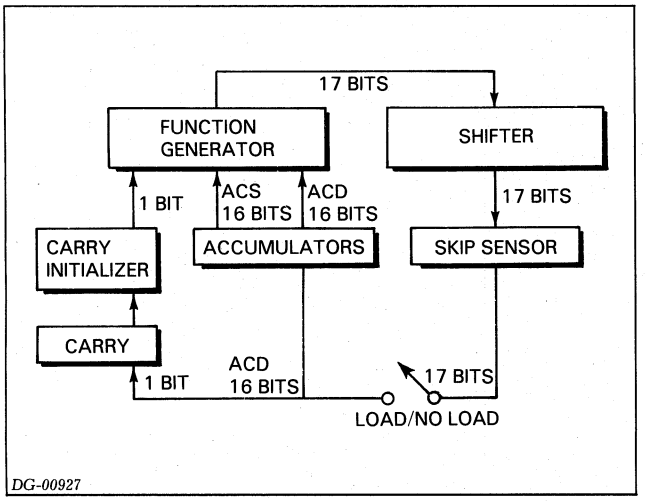


Figure 2.5 Logical organization of the ALU

When an ALC instruction begins execution, it loads the contents of carry and the contents of the accumulator(s) to be processed into the ALU. There are five distinct stages of ALU operation: carry, function, shift operations, skip tests, and load/no-load. We will discuss these stages separately.

Carry

The ALU begins its manipulation of the data by determining a new value for carry. This new value is based upon three things: the old value of carry, bits 10-11 of the ALC instruction, and the ALC instruction being executed. The ALU first determines the effect of the instruction bits 10-11 on the old value of carry. Table 2.10 shows each of the mnemonics that can be appended to the instruction mnemonic, the value of bits 10-11 for each choice, and the action each one takes.

Symbols	Value	Operation
[c] omitted	00	Leave carry unchanged.
[c]=**Z**	01	Initialize carry to 0.
[c]=**O**	10	Initialize carry to 1.
[c]=**C**	11	Complement carry.

Table 2.10 Selecting the value of carry

Function

The ALU next evaluates the effect of the specific function (bits 5-7) upon the data. For the instructions *Move, AND,* and *Complement* the ALU performs the function on the data word(s) and saves the result. The value of carry is as it was calculated above. For the instructions *Add, Add Complement, Subtract, Negate,* and *Increment* the result of the function's action upon the data word(s) may be larger than 2^{16} - 1. An overflow results. In this situation, the ALU saves the low-order 16 bits of the function result, but it complements the value of carry calculated above.

NOTE: *At this stage of operation, the ALU does not load either the saved value of the function result into the destination accumulator, or the calculated value of carry into carry.*

Shift Operations

Next the ALU performs any specified shift operation on the 17 bits output from the function generator (16 bits of data plus the calculated value of the carry bit). Depending on which shift operation is specified in the instruction, the function generator output can be rotated left or right one bit, or have its bytes swapped. Table 2.11 shows the different shift operations that can be performed, the value of bits 8-9 for each choice, and the action each choice takes. Figure 2.6 shows how each shift operation works.

Symbols	Value	Operation
[sh] omitted	00	Do not shift the result of the ALC operation.
[sh]=**L**	01	Rotate left the 17-bit combination of carry bit and ALC operation result.
[sh]=**R**	10	Rotate right the 17-bit combination of carry bit and ALC operation result.
[sh]=**S**	11	Swap the two 8-bit halves of the ALC operation result without affecting carry bit.

Table 2.11 Selecting the shift operation

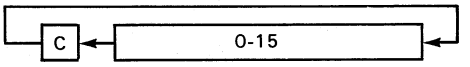
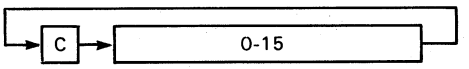
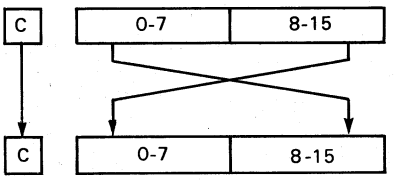
Coded Character	Shifter Operation
L	Left rotate one place. Bit 0 is rotated into the carry position, the carry bit into bit 15. 
R	Right rotate one place. Bit 15 is rotated into the carry position, the carry bit into bit 0. 
S	Swap the halves of the 16-bit result. The carry is not affected. 

Figure 2.6 Effects of shift operations

Skip Tests

The ALU can test the result of the shift operation for one of a variety of conditions, and skip or not skip the next instruction depending upon the result of the test. Table 2.12 shows the tests that can be performed, the value of bits 13-15 for each choice, and the action each choice takes.

Symbol	Value	Operation
[skip] omitted	000	No skip.
[skip]=**SKP**	001	Skip unconditionally.
[skip]=**SZC**	010	Skip if carry bit is zero.
[skip]=**SNC**	011	Skip if carry bit is nonzero.
[skip]=**SZR**	100	Skip if ALC result is zero.
[skip]=**SNR**	101	Skip if ALC result is nonzero.
[skip]=**SEZ**	110	Skip if either ALC result or carry bit is zero.
[skip]=**SBN**	111	Skip if both ALC result and carry bit are nonzero.

Table 2.12 Selecting skip tests

Load/No-Load

If the no-load bit (bit 12) is 0, the ALU loads the result of the shift operation into the destination accumulator, and loads the new value of carry into carry. If the no-load bit is 1, then the ALU does not load the result of the shift operation into the destination accumulator, and does not load the new value of carry into carry, but all other operations, such as skip tests, take place. This no-load option is particularly convenient to use when you want to test for some condition without destroying the contents of the destination accumulator. Table 2.13 shows how to code the load/no-load operation.

Symbol	Value	Operation
# omitted	0	Load the result of the shift operation into ACD.
#	1	Do not load the ALC operation result into ACD; restore carry bit to value it had before shifting.

Table 2.13 Codes for the load/no load option

NOTE: *These instructions must not have both the* No-Load *and the* Never-Skip *options specified at the same time. This bit combination is used to specify other non-ALC instructions.*

The Stack

The stack is a series of consecutive locations in memory. In their simplest form, stack instructions add items in sequential order to the top of the stack and retrieve them in the reverse order. Several stack areas may be defined by the program, but only one stack may be in use at any time. The ECLIPSE S/140 uses the push-down stack concept to provide easily accessible temporary storage of data, variables, return addresses, and the like.

The simplest use of the stack is for temporary storage of the contents of up to four accumulators, which can be stored or retrieved with one instruction. More commonly, the stack is used to store a *return block* which greatly simplifies the process of entering and returning from subroutines.

The return block can take several forms, but it usually consists of five words: the contents of the four accumulators in the first four words, and the program counter and carry in the last word pushed.

Three parameters define a stack: (1) the lower limit, or starting location; (2) the upper limit, or stack limit; and (3) the present top of the stack, or stack pointer. The lower and upper limits define the area in memory which is reserved for the stack, and the stack pointer defines the location of the last word placed onto the stack (or the next word available from the stack). A diagram of a stack area is shown in Figure 2.7.

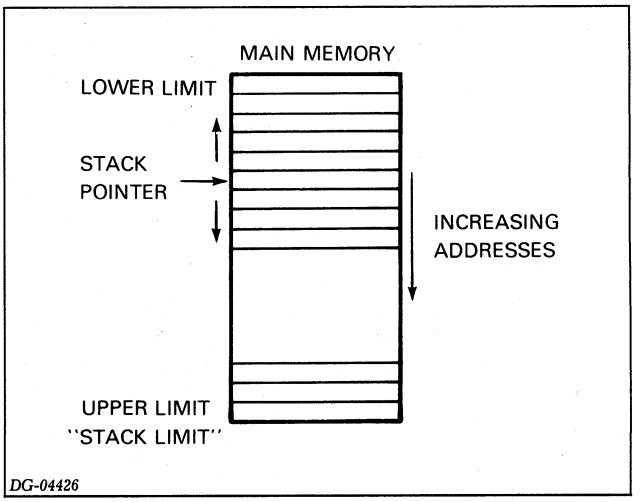


Figure 2.7 The stack area

To use the stack, define the upper and lower limits, then use the stack instructions to put items on *(push onto)* or remove items from *(pop off)* the top of the stack. It is not necessary to keep track of the location of the top of the stack. This is done automatically by one of the stack control words (stack pointer).

Stack Control Words

The stack control words are:

- Stack pointer
- Frame pointer
- Stack limit
- Stack fault address

The locations and uses of the stack control words are discussed in detail below.

Stack Pointer

The stack pointer contains the address of the current top of the stack. As you do push or pop operations, the value of the stack pointer changes so that it always points to the top word of the stack. A push operation increments the stack pointer contents by one, then stores the word you want to push in the new location specified by the stack pointer. A pop operation takes the contents of the word addressed by the stack pointer and loads them into a register, then decrements the stack pointer value by 1.

When you set up the stack, you usually set the value of the stack pointer to be one less than the address of the first stack word.

Location 40_8 contains the current value of the stack pointer.

Frame Pointer

Unlike the stack pointer, the frame pointer does not change its value when push and pop operations occur. If you set the frame pointer to contain the original value of the stack pointer, you have a useful reference to the first stack location.

The *Save* and *Return* instructions use the frame pointer to save the value of the stack pointer when entering or exiting subroutines. Since the frame pointer remains unchanged, it allows you to call a subroutine, perform some operation, then return to the calling program without destroying the value of the stack pointer. This means you can restore the original state of the calling program when you return from the subroutine call.

Location 41_8 contains the value of the frame pointer.

Stack Limit

The stack limit contains the upper limit of the stack area. Each push operation compares the stack pointer with the stack limit to check if there is space enough to allow the push. If the stack pointer is greater than the stack limit, then you have exceeded the size of the stack (overflow condition). For more information, see the next section on "Stack Protection."

Location 42_8 contains the value of the stack limit.

Stack Fault Address

If you cause an overflow or underflow, control transfers to the stack fault routine. For more information, see the next section on "Stack Protection."

Location 43_8 contains the (possibly indirect) address of the stack fault routine.

Stack Protection

You can enable protection for two stack error conditions: *overflow* and *underflow*.

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

Overflow protection is provided by the stack limit. If a stack instruction pushes data onto the stack beyond the stack limit, a return block is pushed onto the stack, and control is transferred to the stack fault handler. To disable overflow protection, the stack limit should be set to 177777_8.

To be meaningful, the stack limit must be 10 to 23 addresses lower than the last word you reserve for the stack, because stack overflow is detected only at the end of a push operation (except in the case of the *Save* and the *Modify Stack Pointer* instructions - see details in Chapter 4). Thus, it is possible to push a 5- to 18-word return block starting at the stack limit. Stack overflow will not be sensed until the last word of the return block is pushed. After the last word is pushed, stack overflow will be detected, and another 5-word return block will be pushed by the stack overflow mechanism before control is transferred to the stack fault routine. Depending on the size of the initial return block (from the normal 5 words up to the 18 words used by the floating point instruction set), the potential overflow can be 10 to 23 words long.

Stack underflow occurs when a program pops data from the area below that allocated for the stack (i.e., pops more words off than were pushed on). If this occurs, the program will be operating with incorrect and unpredictable information. Furthermore, it is possible that the program will push data into the underflow area, overwriting data or instructions.

For underflow protection to be enabled, the area allocated to the stack must begin at location 401_8 and the stack pointer must be initialized to 400_8. If the stack pointer is less than 400_8 after a pop operation, an underflow condition exists and a stack fault occurs.

Underflow protection can be disabled in two ways:

- Start the stack at a location greater than 401_8. A stack fault will not occur then unless the program underflows the stack and then continues to pop words off the stack until the stack pointer is less than 400_8. Note that this does not completely disable underflow protection - it is always possible to pop enough words off the stack to underflow it.
- Set bit 0 of both the stack pointer and the stack limit to 1. If this is done, all or a portion of the stack may reside in page zero (locations $0\text{-}377_8$), or the stack may

underflow into page zero, without interference from the stack underflow mechanism.

Stack Overflow Protection

The *Save* and the *Modify Stack Pointer* instructions check for overflow before executing. For every other instruction that pushes data onto the stack, a check is made for overflow after the execution of the instruction. In both cases, the stack pointer and stack limit are treated as unsigned 16-bit integers and compared. If overflow has occurred, the processor:

- Sets bit 0 of the stack pointer to 0.
- Sets bit 0 of the stack limit to 1.
- Pushes a return block onto the stack.
- Executes a *jump indirect* to the stack fault address.

Bit 0 of the stack pointer and stack limit are set as indicated so that the stack limit will (temporarily) be larger than the stack pointer. In this way, the return block pushed by the overflow mechanism itself will not be interpreted as yet another overflow fault, causing a loop condition. The program counter in the return block points to the instruction immediately following the stack instruction that caused the fault.

Stack Underflow Protection

After every operation that pops data off the stack, a check is made for underflow. If the stack pointer is less than 400_8, and bit 0 of the stack limit is 0, a stack underflow condition exists. In that case, the processor:

- Sets the stack pointer equal to the stack limit.
- Sets bit 0 of the stack pointer to 0.
- Sets bit 0 of the stack limit to 1.
- Pushes a return block onto the stack.
- Executes a *jump indirect* to the stack fault address.

Bit 0 of the stack pointer and stack limit are set as indicated so that the stack limit will (temporarily) be larger than the stack pointer. In this way, the return block being pushed onto the stack by the underflow mechanism (starting at the stack limit) will not cause an overflow fault. The program counter in the return block points to the instruction immediately following the stack instruction that caused the fault.

Stack Fault Handler

The stack fault handler (created by the programmer) determines the nature of the fault. It also resets the appropriate values, and takes any other appropriate action, such as allocating more stack space or terminating the program. Note that the stack fault handler must reset bit 0 of the stack pointer and stack limit to their original values.

Initializing the Stack Control Words

Initialize the stack control words before performing the first operation on the stack. The initialization rules are discussed below, while Figures 2.8 through 2.10 illustrate protected and unprotected stack areas. Figure 2.8 shows a stack area of 50_8 words with underflow protection; Figure 2.9 shows a stack area of 50_8 words in page zero with overflow protection; and Figure 2.10 shows a stack area of 100_8 words with no protection.

Initialize the stack pointer to the beginning address of the stack minus one. If you wish stack underflow protection, initialize the stack pointer to 400_8 and start the stack at 401_8. Otherwise, start the stack at a location greater than 401_8. To place all or a portion of the stack in page zero, or to disable underflow protection, set bit 0 of the stack pointer and the stack limit to 1.

Initialize the stack limit to a value greater than the stack pointer. If you wish stack overflow protection, initialize the stack limit to the last allocated stack address minus at least 10_{10}. Otherwise, initialize the stack limit to 77777_8. To place all or a portion of the stack in page zero, set bit 0 of the stack pointer and the stack limit to 1.

Initialize the stack fault address to an address (determined by the programmer) that contains the routine to handle stack overflow or underflow. Bit 0 may be set to 1 to indicate an indirect address.

The frame pointer will have no meaning until the first use of the *Save* instruction.

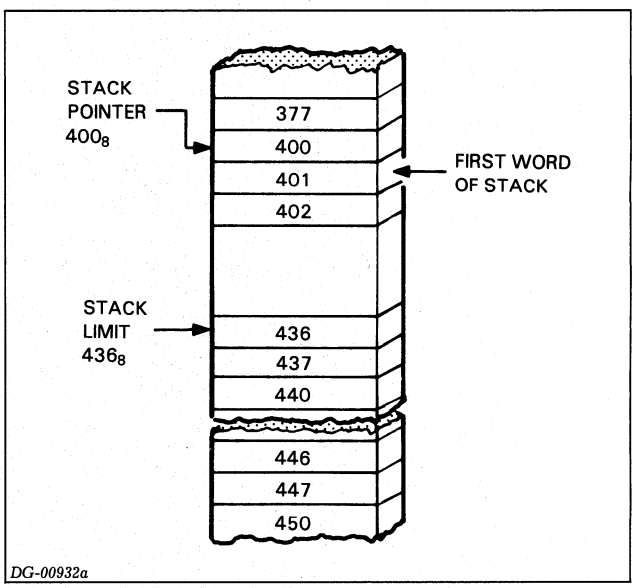


Figure 2.8 Underflow protected stack area

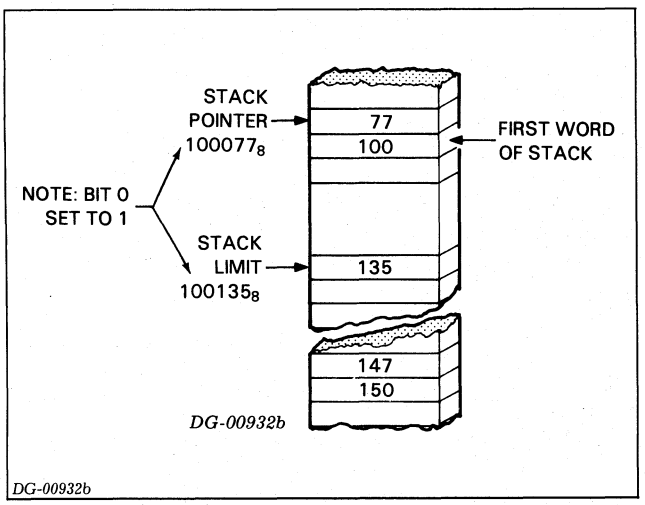


Figure 2.9 Overflow protected page zero stack area

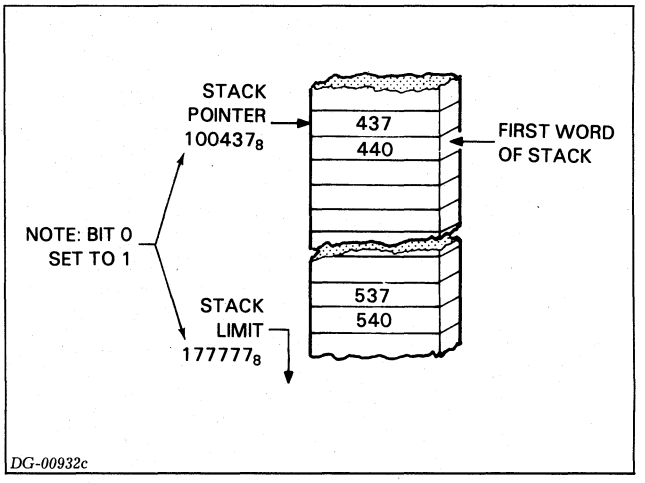


Figure 2.10 Unprotected stack area

Stack Instructions

The instructions that control use of the stack are listed in Table 2.14.

Mnem	Instructions	Action
FPOP	Pop Floating Point State	Pops an 18-word floating point return block off the stack.
FPSH	Push Floating Point State	Pushes an 18-word floating point return block onto the stack.
MSP	Modify Stack Pointer	Changes the value of the stack pointer and checks for overflow.
POP	Pop Multiple Accumulators	Pops 1 to 4 words off the stack and places them in the indicated accumulators.
POPB	Pop Block	Returns control from a *System Call* routine or an I/O interrupt handler that does not use the stack change facility of the *Vector* instruction.
POPJ	Pop PC And Jump	Pops the top word off the stack and places it in the program counter.
PSH	Push Multiple Accumulators	Pushes the contents of 1 to 4 accumulators on the stack.
PSHJ	Push Jump	Pushes the address of the next sequential instruction on the stack and places an effective address into the program counter.
PSHR	Push Return Address	Pushes the address of the PC, plus 2, onto the stack.
RSTR	Restore	Returns control from certain types of I/O interrupts.
RTN	Return	Returns control from subroutines that issue a *Save* instruction at their entry points.
SAVE	Save	Saves the information required by the *Return* instruction.
SYC	System Call	Pushes a return block and indirectly places the address of the *System Call* handler in the program counter.
VCT	Vector on Interrupting Device Code	Performs various interrupt functions. See the I/O section in this chapter.

Table 2.14 Stack instructions

Reserved Storage Locations

The following are reserved storage locations in the ECLIPSE S/140. The CPU uses them for specific functions; you should not use them during normal operations.

The addresses, names, and functions of these locations are given below. The notation *indirectable* means that bit 0 may be set to indicate that this is an indirect address.

The locations shown in Table 2.15 are in unmapped logical address space.

Loc	Name	Function
0	I/O RETURN ADDRESS	Return address from I/O interrupt; first instruction of Auto-restart routine.
1	I/O HANDLER ADDRESS	Address of the I/O interrupt handler (indirectable).
2	SC HANDLER ADDRESS	Address of the *System Call* instruction handler (indirectable).
3	PF HANDLER ADDRESS	Address of the protection fault handler (indirectable).

Table 2.15 Reserved locations in unmapped logical address space

The locations shown in Table 2.16 may be in unmapped logical address space or in Map A or Map B logical address space. They are used by the **VCT** instruction.

Loc	Name	Function
4	VECTOR STACK POINTER	Address of the start of the vector stack (not indirectable).
5	CURRENT MASK	Current interrupt priority mask.
6	VECTOR STACK LIMIT	Address of the last normally usable location in the vector stack (not indirectable).
7	VECTOR STACK FAULT ADDRESS	Address of the vector stack fault handler (indirectable).

Table 2.16 Reserved locations used by the *Vector* instruction

The locations shown in Table 2.17 are in the same address space as the instructions using them.

Loc	Name	Function
20-27	AUTO-INC0 through AUTO-INC7	Auto-incrementing locations.
30-37	AUTO-DEC0 through AUTO-DEC7	Auto-decrementing locations.
40	STACK POINTER	Address of the top of the stack (not indirectable).
41	FRAME POINTER	Address of the frame reference within the stack (not indirectable).
42	STACK LIMIT	Address of the last normally usable location in the stack (not indirectable).
43	STACK FAULT ADDRESS	Address of the stack fault handler (indirectable).
44	XOP ORIGIN ADDRESS	Address of the start of XOP (not indirectable).
45	FLOATING POINT FAULT ADDRESS	Address of the floating point fault handler (indirectable).
46- 47	—	Reserved for future use.

Table 2.17 Reserved locations in the same address space as the instructions using them

Program Execution

A 15-bit register called the *program counter* always contains the address of the instruction currently being executed. During *sequential operation,* the program counter is incremented by one after each instruction. It can normally address the complete logical address space, i.e., 0 through 77777_8, inclusive, a total of 32,768 word locations. The address after 77777_8 is 0, and no indication is given when the counter rolls from 77777_8 to 0 in the course of sequential processing.

Program Flow Alteration

You can alter the program flow from sequential operation in two ways. Jump instructions alter the program flow by inserting a new value into the program counter. Conditional skip instruction alter the program flow by incrementing the program counter an extra time if a specified test condition is true. In either case, sequential operation continues with the instruction addressed by the updated value of the program counter. Figure 2.11 illustrates the effects of these instructions.

NOTE: *Do not use a conditional skip immediately before a 2-word instruction. The conditional instruction causes a 1-word skip, which results in an attempt to execute the second word of the instruction as a 1-word instruction.*

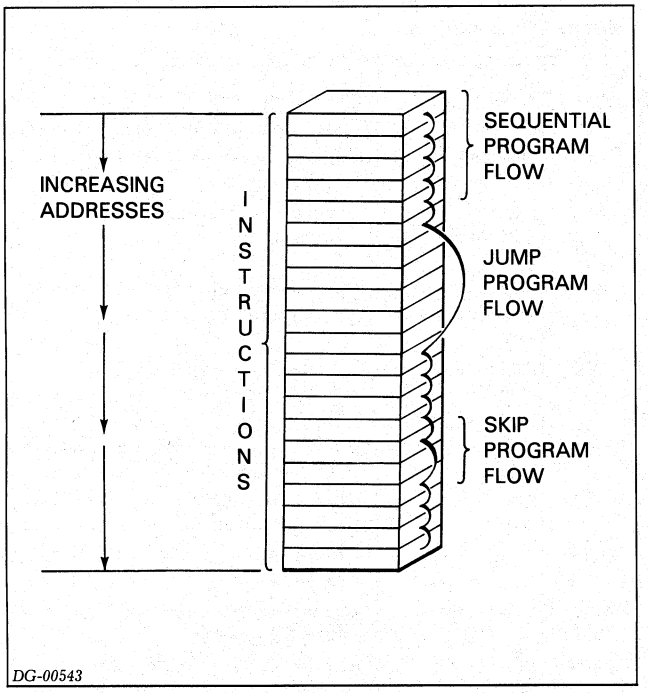


Figure 2.11 Program flow alteration

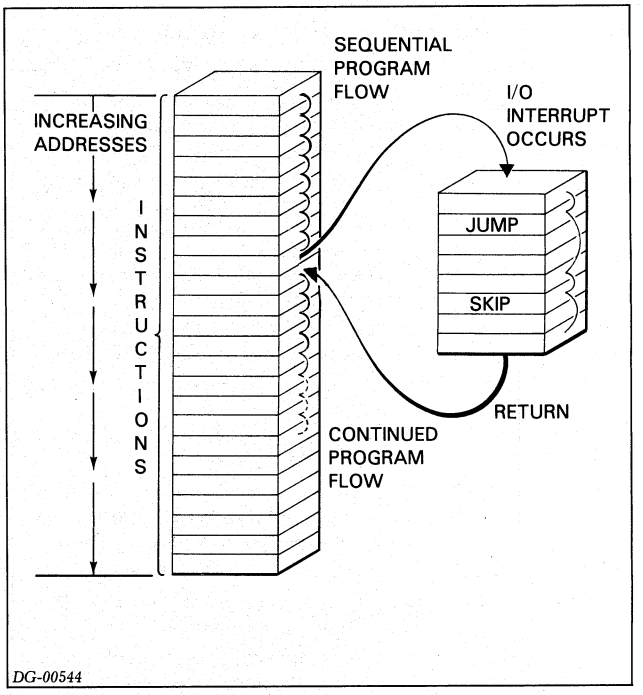


Figure 2.12 Program flow interruption

Program Flow Interruption

The normal flow of a program may be interrupted by external or exceptional internal conditions, such as I/O interrupts or MMPU faults. When this occurs, the contents of the program counter are saved, so that after the interrupt is serviced, control will return to the right place. The address of the starting instruction for the proper fault or interrupt handler is then placed in the program counter and sequential operation continues within that program. When the fault or interrupt handler has serviced the interrupt, control is returned to the interrupted program at the saved address. Figure 2.12 is a diagram of the effect of an interrupt on normal program flow.

Program Flow Alteration Instructions

Program flow alteration and conditional instructions are shown in Tables 2.18 through 2.21. In Table 2.18, several instructions have both short and long forms. The sixteen bit short form directly specifies either a memory address from 0 to 255 or a small area in memory surrrounding the present value of the program counter or an accumulator. The thirty-two bit long form directly specifies any address from 0 to 77777_8.

Table 2.19 summarizes the skip instructions that test condition codes in the floating point status register. Table 2.20 summarizes the condition tests available for the **SKP** *[t]* instruction. (This instruction tests the condition codes of a peripheral device, the power-fail monitor or the interrupt system.) And Table 2.21 summarizes the *skip* options of the ALC instructions.

Mnem	Instructions	Action
CLM	Compare To Limits	Compares a signed integer with two other numbers and skips if first integer is between the other two.
DSPA	Dispatch	Compares a signed integer with two other numbers and continues sequential execution if the integer is not between the others; otherwise, uses the integer as an index into a table and places indexed value in the program counter.
DSZ EDSZ	Decrement And Skip If Zero	Decrements the addressed word, then skips if the decremented value is zero.
ISZ EISZ	Increment And Skip If Zero	Increments the addressed word, then skips if the incremented value is zero.
JMP EJMP	Jump	Places an effective address in the program counter.
JSR EJSR	Jump To Subroutine	Increments program counter and stores incremented value in AC3; then places a new address in the program counter.
POPB	Pop Block	Pops a return block off of the stack.
POPJ	Pop PC And Jump	Pops the top word off the stack and places it in the program counter.
PSHJ	Push	Pushes the address of the next sequential instruction onto the stack and places a new address in the program counter.
RSTR	Restore	Returns control from I/O interrupt handlers that use the stack change facility of the **VCT** instruction.
RTN	Return	Returns control from a subroutine entered via *Save* instruction.
SGE	Skip If ACS Greater Than Or Equal To ACD	Compares two signed integers in two accumulators and skips if the first is greater than or equal to the second.
SGT	Skip If ACS Greater Than ACD	Compares two signed integers in accumulators; skips if first is greater than the second.
SKP*[t]*	I/O Skip	Skips if the I/O condition *t* is true.
SNB	Skip On Nonzero Bit	References a single bit in memory via bit pointer; skips if bit is 1.
SYC SCL SVC	System Call	Turns the MAP off if on. Pushes a return block onto the stack places address of *System Call* handler in program counter.
SZB	Skip On Zero Bit	References a single bit in memory via bit pointer; skips if bit is 0.
SZBO	Skip On Zero Bit, Set To 1	References a single bit in memory via bit pointer; skips if bit is 0 and also sets the bit to 1.
VCT	Vector On Interrupting Device Code	Identifies highest priority interrupt; passes control through a table to a handler routine for device.
XOP XOP1	Extended Operation	Pushes a return block onto the stack, indexes into the XOP table and transfers control to another procedure.
XCT	Execute	Executes contents of an accumulator as an instruction.

Table 2.18 Program flow alteration and conditional instructions

Mnem	Instructions	Action
FNS	No Skip	The next sequential word is executed.
FSA	Skip Always	The next sequential instruction is skipped.
FSEQ	Skip On Zero	Skips the next sequential word if the Z flag in the FPSR is 1.
FSGE	Skip On Greater Than Or Equal To Zero	Skips the next sequential word if the N flag of the FPSR is 0.
FSGT	Skip On Greater Than Or Equal To Zero	Skips the next sequential word if both the Z and N flags of the FPSR are 0.
FSLE	Skip On Less Than Or Equal To Zero	Skips the next sequential word if either the Z flag or the N flag of the FPSR is 1.
FSLT	Skip On Less Than Zero	Skips the next sequential word if the N flag of the FPSR is 1.
FSND	Skip On No Zero Divide	Skips the next sequential word if the divide by zero (DVZ) flag of the FPSR is 0.
FSNE	Skip On Non-Zero	Skips the next sequential word if the Z flag of the FPSR is 0.
FSNER	Skip On No Error	Skips the next sequential word if bits 1-4 of the FPSR are all 0.
FSNM	Skip On No Mantissa Overflow	Skips the next sequential word if the mantissa overflow (MOF) flag of the FPSR is 0.
FSNO	Skip On No Overflow	Skips the next sequential word if the overflow (OVF) flag of the FPSR is 0.
FSNOD	Skip On No Overflow And No Zero Divide	Skips the next sequential word if both the overflow (OVF) flag and the divide by zero (DVZ) flag of the FPSR are 0.
FSNU	Skip On No Underflow	Skips the next sequential word if the underflow (UNF) flag of the FPSR is 0.
FSNUD	Skip On No Underflow And No Zero Divide	Skips the next sequential word if both the underflow (UNF) flag and the divide by zero (DVZ) flag of the FPSR are 0.
FSNUO	Skip On No Underflow And No Overflow	Skips the next sequential word if both the underflow (UNF) flag and the overflow (OVF) flag of the FPSR are 0.

Table 2.19 Floating point skip instructions

Symbol	Value	Test
[t]=**BN**	00	Tests Busy flag for nonzero.
[t]=**BZ**	01	Tests Busy flag for zero.
[t]=**DN**	10	Tests Done flag for nonzero.
[t]=**DZ**	11	Tests Done flag for zero.

Table 2.20 **SKP***[t]* condition tests

Symbol	Value	Operation
[skip] omitted	000	No skip.
[skip]=**SKP**	001	Skip unconditionally.
[skip]=**SZC**	010	Skip if Carry bit is zero.
[skip]=**SNC**	011	Skip if Carry bit is nonzero.
[skip]=**SZR**	100	Skip if ALC result is zero.
[skip]=**SNR**	101	Skip if ALC result is nonzero.
[skip]=**SEZ**	110	Skip if either ALC result or Carry bit is zero.
[skip]=**SBN**	111	Skip if both ALC result and Carry bit are nonzero.

Table 2.21 ALC *skip* options

Extended Operation Feature

The extended operation feature (XOP) provides an efficient method of transferring control to and from procedures. It enables the user to transfer control to any one of 48 procedure entry points.

Extended Operation Instructions

There are two extended operation instructions in the ECLIPSE S/140 instruction set. They are shown in Table 2.22.

Mnem	Instructions	Action
XOP	Extended Operation	Pushes a return block on the stack; places the address of the specified accumulators into AC2 and AC3; and transfers control to one of thirty-two other procedures via the XOP table.
XOP1	Extended Operation	Same as **XOP** except that 32 is added to the entry number before entering the XOP table, and only 16 table entries can be specified.

Table 2.22 Extended operation instructions

Input/Output

This section contains descriptions of the Input/Output capabilities in the ECLIPSE S/140. We discuss the general operation of the I/O system, interrupts, and vectoring.

The ECLIPSE S/140 has a 6-bit device selection network, corresponding to bits 10-15 in the I/O instruction format. The devices are connected to this network in such a way that each device will only respond to commands sent with its own device code. With a 6-bit device code, 64 separate devices can be individually controlled. Certain specific codes are reserved for the CPU and certain processor options; the remainder reference I/O devices. The assembler recognizes mnemonics for those devices assigned a code by Data General. A complete list of these is provided in Appendix B of this manual.

See the *Programmer's Reference Manual - Peripherals* (DGC No. 014-000632 for details about programming specific devices in the I/O system.

Busy and Done Flags

Most I/O devices are controlled through the manipulation of Busy and Done flags. Flag values are changed through the use of optional flag command mnemonics. The effects of the flag commands are device dependent.

Programmed I/O

Programmed I/O transfers data one word at a time under direct program control. For slow devices, such as teletypes, which transfer one character at a time and require an immediate echo, programmed I/O is the fastest method of I/O operation.

For faster devices, programmed I/O has several disadvantages. Several instructions are required for the transfer of each word, and other CPU operations must wait for the transfer to be completed. Because data must be transferred to or from an accumulator, an additional step is required if the data must be stored in or retrieved from memory.

Data Channel I/O

Data channel I/O permits data transfer in blocks of words, with program control necessary only at the start of the operation. The CPU stops during each word transfer; but the transfer is made directly to or from memory, so no additional steps are required. Data channel I/O very efficiently transfers large blocks of data between memory and a fast I/O device.

Dual data channel transfer rates are 1.4 Mbytes per second normal input, 1.1 Mbytes per second normal output; and 2.0 Mbytes per second fast input, 1.4 Mbytes per second fast output.

At the fast rates, the CPU is effectively stopped. At normal rates, however, processing continues data between transfers.

Data channel devices are controlled in three phases. Phase I specifies the starting location in memory for the first word to be transferred. Phase II loads the two's complement of the number of words to be transferred into the machine. These two phases are performed with programmed I/O instructions. Phase III issues a flag command. Once a flag command is issued, data transfer takes place when both the data channel device and the processor are ready. No further program control is required.

When a data channel device is ready to send or receive data, it issues a data channel request to the processor. The processor synchronizes any requests that are coming in. At certain specified points, the CPU pauses to honor all previously synchronized requests. When a request is honored, a word is transferred directly via the data channel between the device and memory without specific action by the program.

All requests are honored according to the relative position of the requesting device on the I/O bus. Data channel service begins with the device that is physically closest on the bus. The next closest device is serviced next, and so on, until all requests have been honored. New requests are synchronized concurrently with the servicing of older requests. If a device continually requests data channel service, it prevents all devices further out on the bus from gaining access to the channel.

For more information on the data channel, see the *Programmer's Reference Manual - Peripherals* (DGC No. 014-000632) and the *User's Manual - Interface Designer's Reference* (DGC No. 014-000629).

I/O Interrupts

The I/O interrupt system in the ECLIPSE S/140 manages programmed I/O by permitting the program to ignore I/O devices until one requires service. After handling all data channel requests, the processor completes execution of any incomplete instruction, services any further data channel requests that were synchronized while the instruction was executing, then services outstanding I/O interrupt requests. When all requests have been serviced, program execution continues.

The following definitions will assist your understanding of system interrupts.

Interrupt request line — This line is the common connection between all I/O devices and the computer. In general, an I/O device places a request on the interrupt request line while it sets Busy to 0 and Done to 1. That is, it signals that it is ready to send or receive data. The program must use a separate means to determine which device is requesting an interrupt.

Interrupt On flag — This CPU flag controls the status of the interrupt system. When the flag is set to 1, the CPU will respond to and process interrupts. If the flag is set to 0, the CPU will not examine the interrupt request line.

Priority mask — This set of bits controls the priority interrupt system. Every I/O device is connected to at least one of the sixteen bits in the priority mask. When a mask bit is set to 1, the devices connected to it cannot place a request on the interrupt request line. The devices can set their Busy flags to 0 and their Done flags to 1. Since the program controls the mask, devices are at times inhibited in order to conform to a priority system.

Base level — In this program state, no I/O devices are inhibited (all mask bits are 0) and no interrupts are processed. User program execution takes place here.

Non-base level — In this program state, some I/O devices are masked or interrupts are processed. Interrupt handlers operate in this state.

In the next section we will discuss interrupts. We will first consider interrupts without a priority system; and then, interrupts within a priority system.

Processing Interrupts With No Priority System

When a device completes an operation and is ready to send or receive more data, it sets the Busy flag to 0 and the Done flag to 1. The device bit in the priority mask is 0, so the device places a request on the interrupt request line. At the next opportunity, the interrupt is serviced.

To service an interrupt, the CPU first sets the Interrupt On flag to 0 to prevent interruption of the first part of the interrupt service routine. Second, it disables the user map. Then the CPU places the contents of the updated program counter into physical memory location 0 and jumps indirect via location 1, where it expects to find the address (direct or indirect) of the interrupt service routine.

The interrupt service routine (supplied by the user) must save any accumulators and the carry bit if they are used, and determine which device requested the interrupt. Then the service routine tends to the device.

The service routine can use the *I/O Skip* or *Interrupt Acknowledge* instructions to identify the device requesting the interrupt. Or it can use the *Vector on Interrupting Device Code* instruction to save the return information and identify the interrupting device.

The *Interrupt Acknowledge* instruction returns the 6-bit device code of the device requesting the interrupt. The *Vector* instruction, in addition to saving return information on the stack, performs an *Interrupt Acknowledge* instruction and uses the code returned as an index into a table of addresses. These addresses point to the beginnings of the various device service routines.

After servicing the device, the interrupt routine should restore the saved values of the accumulators and the carry bit, set the Interrupt On flag to 1, and return to the interrupted program. The *Interrupt Enable* instruction sets the Interrupt On flag to 1, and allows the processor to execute one more instruction before allowing the next interrupt.

This next instruction should return control to the interrupted program. Since the updated value of the program counter was placed in location 0 by the CPU at the start of the interrupt service routine, a *jump indirect,* via location 0, returns control to the proper location in the interrupted program.

Processing Interrupts With a Priority System

Two factors explain why a system of priorities for interrupts is necessary. First, if the Interrupt On flag remains 0 throughout the interrupt service routine, the CPU cannot be interrupted while an I/O device is being serviced. All other devices, therefore, must wait until the first device is finished. Second, if the Interrupt On flag is returned to 1 after the initial portion of the service routine, any I/O device can interrupt the servicing of any other I/O device. This might be reasonable for some devices, but it is not for others. Therefore, a system of interrupt priorities is needed to permit some devices to interrupt certain others without disrupting the orderly processing of data.

A rudimentary priority system will result from keeping the Interrupt On flag 0 throughout the service routine. The priority of the I/O devices is then determined either by the order in which the *I/O Skip* instructions poll the I/O devices, or (using the *Interrupt Acknowledge* or *Vector* instructions) by the physical location on the I/O bus of devices requesting an interrupt. Both methods, however, are very inflexible.

The ECLIPSE S/140 has the hardware and instructions for a flexible and efficient priority system, with up to sixteen levels of priority interrupts. The interrupt service routine controls the priority system, and changes the priorities of various devices as necessary.

Using the Mask Out Instruction: To set up a system of priorities, place a *Mask Out* instruction in the interrupt service routine for each device. This instruction changes the priority mask and prevents certain devices from requesting interrupts. If particular bits in the priority mask are changed to 1, the devices are *masked out*. Devices controlled by those bits are disabled from requesting interrupts. Priority mask bits that correspond to devices not masked out, are set to 0.

If the priority interrupt system is called in each interrupt service routine, devices that do not merit an interrupt will be masked out. The process changes each time a different device is serviced, resulting in a system of priorities. The device with the highest priority will be able to interrupt all other devices; and the lowest priority device will be interruptible by other devices.

Devices which operate at roughly the same speed are controlled by the same bit in the mask. Appendix B lists the mask bit assignments along with the device code assignments. Although the bit assignments are fixed, the priorities are set by the programmer to fit the situation and are dynamically adjustable.

Using Interrupt Handlers and Service Routines: The initial portions of a multiple priority level interrupt handler may be damaged if the routine is interrupted. To prevent this from occurring, the processor has automatically set the Interrupt On flag set to 0. After receiving control, the interrupt handler must save return information and store it in a unique place to prevent its being overwritten by data from another interrupt.

Next, choose a service routine that will save the current priority mask and establish a new one.

Then, use the *Interrupt Enable* instruction to set the Interrupt On flag to 1. This permits those devices not restricted by the priority mask to interrupt if necessary.

After servicing the interrupt, the interrupt service routine should:

- Disable the interrupt system.
- Reset the priority mask to the condition it was in when the routine was entered.
- Restore the accumulators and the carry bit.
- Enable the interrupt system.
- Return control to the interrupted program.

Making Stack Changes: The interrupt handler uses a stack. Instead of working with the user stack, you can define a new stack which is reserved for the interrupt handler. This overcomes the following two problems:

- There may not always be a defined user stack.
- The user stack pointer may rest just below the stack limit. The interrupt handler would then overflow the user stack.

The stack environment should be changed whenever a transition is made from base level to non-base level or vice versa.

If an interrupt is being processed when another interrupt occurs, the stack environment should not be changed, since this has already been done for the first interrupt. If desired, return information to permit an easy return to processing the first interrupt can be pushed onto the new stack before the second interrupt is processed.

The *Vector* instruction handles stack changes by changing modes to accommodate different situations. We discuss the uses for this instruction in the next section.

Using the Vector Instruction: The *Vector on Interrupting Device Code* instruction simplifies the design of an interrupt handler by streamlining numerous steps into one instruction.

The *Vector* instruction contains five modes, each suited to a different circumstance.

The simplest mode, similar to the *Interrupt Acknowledge* instruction, executes rapidly and does not save information about the processor state at the time of the interrupt.

The most complex mode saves information on the state of the machine upon interruption, stores the user stack parameters, creates a new stack and resets the priority mask. This mode executes more slowly than the simpler mode described above.

To choose the correct mode, you must weigh the importance of such capabilities as saving the machine state, creating a separate vector stack, and changing the priority mask, against the time added on to an interrupt. You are not committed to one mode throughout the interrupt handler.

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

Modes B through E Each mode creates 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 Use these modes only when operating at base level (not while interrupts are being processed). They create a new vector stack. The interrupt handler stores the (old) user stack parameters in it. Once the vector stack has been created, do not attempt to recreate it if a new interrupt occurs before the one in progress finishes.

Mode E pushes a return block onto the vector stack to make the return to the first interrupt handler easier.

Modes B and C May be used during non-base level operations (while interrupts are processed). These modes do not create a new stack.

Mode C also pushes a new return block onto the stack.

Chapter 5 gives more details on the *Vector* instruction.

I/O Instructions

Table 2.23 lists the I/O instructions for the ECLIPSE S/140. Some of these instructions have special mnemonics which can be used in place of the standard mnemonics. Note that the mnemonics for controlling the states of flags connot be appended to these special instruction mnemonics.

If you want to alter the state of the Interrupt On flag while performing a *Mask Out* instruction, you must use the full mnemonic:

DOB*f* *ac*,**CPU**

instead of the special mnemonic:

MSKO *ac*

In this example, the special mnemonic sets bits 8 and 9 to 00.

Mnem	Instructions	Action
DIA	Data In A	Transfers data from the A buffer of an I/O device to an accumulator.
DIB	Data In B	Transfers data from the B buffer of an I/O device to an accumulator.
DIC	Data In C	Transfers data from the C buffer of an I/O device to an accumulator.
DOA	Data Out A	Transfers data from an accumulator to the A buffer of an I/O device.
DOB	Data Out B	Transfers data from an accumulator to the B buffer of an I/O device.
DOC	Data Out C	Transfers data from an accumulator to the C buffer of an I/O device.
DOC CPU	Halt	Stops the processor.
DIB CPU	Interrupt Acknowledge	Returns the device code of an interrupting device.
INTDS (NIOC CPU)	Interrupt Disable	Sets Interrupt On flag to 0.
INTEN (NIOS CPU)	Interrupt Enable	Sets Interrupt On flag to 1.
DIC CPU	Reset	Sets all Busy and Done flags and the priority mask to 0.
DOB CPU	Mask Out	Changes the priority mask.
NIO	No I/O Transfer	Changes a flag without causing any other effect.
DIA CPU	Read Switches	Places the contents of the console data switches into an accumulator.
SKP	I/O Skip	Tests a flag and skips the next sequential word if the test condition is true.
SKP CPU	CPU Skip	Tests the Interrupt On or Power Fail flag and skips the next sequential word if the test condition is true.

Table 2.23 I/O instructions

Table 2.24 summarizes the flag commands issued where the optional mnemonics are used in I/O instructions.

Mnem	Instructions	Action
[f] omitted	00	No operation.
[f]=**S**	01	Issues a Start command to the device.
[f]=**C**	10	Issues a Clear command to the device.
[f]=**P**	11	Issues a Pulse command to the device

Table 2.24 Flag command functions

When an I/O instruction employs a special mnemonic, the device Busy and Done flags are tested for the conditions described earlier in Table 2.20.

Table 2.25 applies to I/O instructions using the device code mnemonic **CPU** (device code 77_8). These instructions operate on the Interrupt On and Power Fail flags, rather than testing the Busy and Done flags.

Mnem	Instructions	Action
[f] omitted	00	Does not alter the Interrupt On flag.
[f]=**S**	01	Sets Interrupt On flag to 1.
[f]=**C**	10	Clears Interrupt On flag to 0.
[f]=**P**	11	Leaves Interrupt On flag unchanged (used only with **VCT**).
[t]=**BN**	00	Tests Interrupt On flag for nonzero.
[t]=**BZ**	01	Tests Interrupt On flag for zero.
[t]=**DN**	10	Tests Power Fail flag for nonzero.
[t]=**DZ**	11	Tests Power Fail flag for zero.

Table 2.25 Interrupt On and Power Fail flag commands

Basic I/O Devices

The ECLIPSE S/140 includes two basic I/O devices. They are, the real time clock (RTC) and an asynchronous line controller (ALC).

Real Time Clock

The real time clock generates low frequency I/O interrupts. Use these interrupts in programs that must perform time calculations independently of CPU timing. A program can select one of four clock frequencies: 10Hz, 100Hz, 1000Hz, or AC line frequency.

When the real time clock starts, the first program interrupt request can come at any time. After the first interrupt, succeeding interrupts come at the clock frequency, provided that the program always sets Busy to 1 before the clock period expires. After power up or **IORST**, the clock is set to AC line frequency. Line frequency pulses are available immediately, but five seconds must elapse before a steady pulse train is available from the clock for other frequencies.

A single instruction programs the real time clock. Table 2.26 illustrates that instruction.

Mnem	Instructions	Action
DOA	Select RTC Frequency	Selects the RTC interrupt frequency.

Table 2.26 Real time clock instruction

Asynchronous Line Controller

The asynchronous line controller is the communications interface between the ECLIPSE S/140 and its primary terminal. The controllers support communication at selected baud rates from 50 to 19200, in seven-bit codes with program generated parity, or in eight-bit codes with no parity. One or two stop bits may be used with either format. ALC input and ALC output each have unique device codes and are controlled by their own Busy and Done flags.

The asynchronous line controller is set up to transmit and receive 8-bit characters without parity checking. You can send and receive 7-bit characters with even, odd, or mark parity under program control by using the high order bit in the 8-bit character (bit 8 in the AC) as a parity bit. On transmission, the program which drives the asynchronous line controller may calculate and insert the correct parity bit. On reception, the program may calculate and check parity on the received character.

You must also be aware of timing constraints on the receiver portion of the controller. As a character is received, the controller places it into an input character buffer, sets the Done flag to 1, and the Busy flag to 0. If the program controlling the receiver does not transfer the character before the next character is received, the new character overwrites contents of the input character buffer, and the previous character is lost. At 50 baud, the minimum time before the previous character is overwritten is 220 milliseconds; at 19200 baud the minimum time is approximately 521 microseconds.

One instruction programs the asynchronous line input (ALI). The instruction is shown in Table 2.27.

Mnem	Instructions	Action
DIA	Read Input Buffer	Reads a character from the input buffer.

Table 2.27 Asynchronous line input instructions

A single instruction programs the asynchronous line output (ALO). See Table 2.28.

Mnem	Instructions	Action
DOA	Load Output Buffer	Places a character in the output buffer.

Table 2.28 Asynchronous line output instruction

Power Fail/Auto-restart

When power is turned off, the contents of semiconductor memory are lost. The state of the accumulators, the program counter, and the various flags in the CPU and SC memory then are indeterminate. If you have battery backup the power fail facility provides a *fail-soft* capability in the event of unexpected power loss.

In the event of power failure, there is a delay of one to two milliseconds before the processor shuts down. The power fail facility senses the loss of power, sets the Power Fail flag to 1 and requests an interrupt. The interrupt service routine can then use this delay to store the contents of the accumulators, the carry bit, and the current priority mask. The interrupt service routine should also save location 0 (to enable return to the interrupted program), put a *Jump* to the desired restart location in location 0, and then execute a **HALT.** One to two milliseconds is enough time to execute 1000 to 1500 instructions, so there is more than enough time to perform the power fail routine.

As long as the batteries have not been exhausted, (up to one hour for minimum memory configuration) when power is restored, the action taken by the automatic restart portion of the power fail facility depends upon the position of the lock switch on the front panel. If the switch is *not* in the *lock* position, the CPU remains stopped after power is restored. If the switch is in the *lock* position, then after power is restored, the CPU executes the instruction contained in physical location 0, thereby transferring control to the restart procedure.

Power Fail Instructions

The power fail instructions test the state of the power fail flag. They use the device code 77_8. The assembler recognizes the mnemonic **CPU** for this device code.

The power fail facility has no priority mask bit in the priority mask. It responds to the *Interrupt acknowledge* and *Vector* instructions with device code 0.

Power fail has the lowest priority of all devices for the *Interrupt Acknowledge* instruction, but highest priority for the *Vector* instruction.

The power fail instructions are shown in Table 2.29.

Mnem	Instructions	Action
SKPDN, CPU	Skip If Power Fail Flag Is One	If the Power Fail flag is 1 (i.e., power is failing), the next sequential word is skipped.
SKPDZ, CPU	Skip If Power Fail Flag Is Zero	If the Power Fail flag is 0 (i.e., power is not failing), the next sequential word is skipped.

Table 2.29 Power fail instructions

Error Checking and Correction

The Error Checking and Correction (ERCC) facility is designed for applications requiring either a high degree of reliability for the main memory of a system, or a graceful "fail-soft" capability in the event of memory errors. The ERCC facility will detect and correct all single-bit memory errors. If no error occurs, memory cycle time is identical to non-ERCC cycle time.

Every ERCC memory word is twenty-one bits long. These twenty-one bits consist of sixteen data bits followed by five ERCC check bits. Each time the CPU writes data into a location, a hardware encoder constructs a 5-bit check field from the sixteen data bits. Each time the CPU reads data from a memory location, the hardware encoder constructs another five bit check field based on the sixteen bits read from memory and the five ERCC bits written into memory. If that code is all zeroes, no error occurred and the ERCC facility passes the sixteen data bits on to the CPU. Otherwise, an error occurred. The memory pauses while the ERCC facility corrects the single bit, requests an interrupt and passes the corrected data on to the CPU.

ERCC logic can detect and correct all single-bit errors. In the rare event that a multi-bit error occurs, ERCC either detects it and reports it with no correction, or incorrectly interprets it as a single-bit error and complements the bit.

ERCC Instructions

One I/O instruction sets the mode of operation of the ERCC facility. ERCC contains a Done flag which is set to 1 after an error has been detected and the ERCC initiates an interrupt request. Two instructions interrogate ERCC after the detection and correction of an error.

The ERCC facility has no Busy flag and no mask bit in the priority mask. The device code for the ERCC facility is 2. The assembler recognizes the mnemonic **ERCC** for this device code.

ERCC instructions use a specified accumulator to receive data or contain the control information.

Table 2.30 shows the ERCC instructions.

Mnem	Instructions	Action
DOA	Enable ERCC	Enables the ERCC facility according to the setting of bits 13-15 of the specified accumulator.
DIA	Read Memory Fault Address	Returns the low-order bits of the memory location which has produced an error.
DIB	Read Memory Fault Code	Returns a 5-bit error code that tells which bit was in error. Also returns the high-order bits of the memory fault address.

Table 2.30 ERCC instructions

Virtual Console

The virtual console (VC) allows you to interact with the computer through the system terminal connected to the CPU's on-board asynchronous communications interface. Simple commands which you enter on the terminal keyboard allow you to examine and/or modify processor registers or memory locations; start, stop, and continue program execution; and, initiate a program load from a selected device.

On power up, the computer performs a self-test. After a successful completion of the self-test, the following message appears on the terminal:

OK
!000000
!

OK followed by *!000000* indicates that the self-test ran successfully. The digits following the *!* are the contents of the program counter; on power-up, they are all zeroes. The next *!* is the VC prompt; it tells you that the virtual console is ready and at your service.

In addition to power-up, the VC is entered when one of the following occurs:

- A **HALT** instruction is executed.
- The RESET switch on the front console is pressed and the front console is unlocked.
- The BREAK key on the system terminal is pressed, the front console is unlocked, and the CPU is not in a microcode loop.

Under these conditions, the incremented contents of the program counter are typed when the VC is entered. These are followed by the *! VC* prompt. For example, if the program counter was at location 2077 when the VC is entered, the following would be typed:

002077
!

Cells

The VC operates on 'cells'. A cell is either a memory location (memory cell) or an internal register (internal cell) such as an accumulator. Each internal register that the VC can access has an internal cell number. These cell numbers are listed in Table 2.31.

Internal Cell	Internal Register
0-3	The contents of the accumulators AC0 through AC3, respectively.
4	Return address (the contents of the program counter when VC was entered.)
5	Reserved for future use.
6	Reserved for future use.
7	Interrupt enable flag status bit: 0 = Interrupts off 1 = Interrupts on
10	MMPU status bits (when set to 1) before the VC was entered: 0 MAP on before console mode entered 1 Program MAP enable pending 2 I/O protection fault 3 Write protection fault 4 Indirect protection fault 5 Last map fault occurred during single cycle memory reference 6-8 Map select: 000 User A 001 Reserved for future use 010 User B 011 Reserved for future use 100 Data channel A 101 Data channel C 110 Data channel B 011 Data channel D 9 Load Effective Address (LEF) mode enabled 10 I/O protection enabled 11 Write protection enabled 12 Indirect protection enabled 13 User map enable (0=A, 1=B) 14 Data channel map enable 15 Last interrupt occurred in user mode
12	Data switch register: Replaces the conventional console data switches. When the system is in RUN mode (i.e., not in console mode), and a READS instruction is executed, the 16-bit contents of this register are read by the CPU.
	Value of the carry bit.

Table 2.31 Virtual console internal cells

Cell Commands

In order to examine or modify any cell, you must 'open' it. Opening a cell causes its contents to be printed, in octal, on the terminal. To open a cell, use one of the commands listed in Table 2.32. The VC will respond only to octal numbers and upper case letters.

NOTE: *In the table, the term* **'current cell'** *means the last cell that you opened.*

Command	Function
nA	Open the internal cell whose internal cell number is equal to "n" (See Table 2.31).
n/	Open the memory location whose physical address is equal to the octal number "n".
(carriage return)	Close the current cell, and open the next consecutive cell.
(line feed or new line)	Close the current cell, but do not open another.
/	Close the current cell and open the memory cell whose address is equal to the contents of the current memory or internal cell.

Table 2.32 Virtual console cell commands.

When you open a memory cell, the VC interprets the address as a 20-bit physical address. You do not have to type leading zeroes. All you have to type is the physical address in octal representation. For example, if you want to open location 5, type 5/. If you want to examine the top location of a system which contains 2Mbytes of memory type 3777777/.

Once you have opened a cell, you may change its contents by simply typing (in octal) the number whose value is to be placed in the cell. Terminate the expression with a Carriage Return, Line Feed or New Line. Note that if you type Carriage Return the next cell will also be opened. This is convenient when you need to enter data into several consecutive locations.

Function Commands

Table 2.33 lists the VC function commands. All commands must be typed in octal numbers and upper case letters.

Command	Function
P	Starts program execution at the memory location specified by the contents of internal cell number 4 (see Table 2.31).
nR	Issues an I/O Reset, clears the MMPU, and starts program execution at the memory location specified by the octal number "n".
I	Issues an I/O Reset, and clears the MMPU.
nL	Performs a program load from the device whose device code is equal to "n". Bit 0 of "n" is a 0 for a low-speed device, and is a 1 for a high-speed device.
F	Performs a DG field service cassette bootstrap load. (For DGC use only.)
K	Cancels the entire line just typed, and prints a question mark (?).

Table 2.33 Virtual console function commands

The VC uses two commands to start program execution. Typing **P** starts program execution at the location specified by internal cell number 4 (the return address). See Table 2.31, Virtual console internal cells. You can also start program execution by typing *n***R**. In this case, the CPU issues an I/O Reset command, clears the MMPU, and starts program execution at the location specified by the octal number *n*.

Typing **I** causes the CPU to issue an I/O Reset command and clear the MMPU.

Type *n***L** to program load from an I/O device, where *n* is the device code, in octal, of the I/O device to be used. Bit 0 of *n* should be a 1 if the I/O device is high-speed, and a 0 if the I/O device is low-speed. For example, if the program load device is a high-speed 6060 disc drive whose device code is 27, you would type the following:

```
100027L
```

You can perform a Data General field service cassette bootstrap load by typing **F**.

Virtual Console Errors

If you type a character that the VC does not recognize, it will print a ? followed by a New Line. If you wise to cancel an entire line you just entered, type a **K**. In this case the VC will respond with a **?** followed by a New Line.

If you attempt to open a non-existent memory cell, the 16-bit contents of the cell printed in octal on the terminal will be all 1's. You can verify that this location does not exist by entering a new value containing 0's in the cell and then re-opening it. If it still contains all 1's, the location is non-existent.

If you attempt to open a non-existent internal cell, the terminal will print random and meaningless data.

Memory Management and Protection Unit

The ECLIPSE S/140 MMPU provides the hardware necessary to control and use more than 64 Kbytes of physical memory. In addition, the MMPU provides protection functions which help protect the integrity of a large system.

An MMPU unit gives several users access to the resources of the computer by divding the memory space available into blocks assigned to each user. Each time a user accesses memory, the MMPU translates the address the user sees, the *logical address*, to an address the memory sees, the *physical address*. This is all transparent to the user. With software to control the priorities of the MMPU and the CPU, several users can access the computer without being aware of the presence of the others.

For the purposes of this discussion, we define certain words and phrases:

Logical address—The address used by the user in all programming. The logical address space is 32,768 words long and is addressed by a 15-bit address.

Physical address—The address used by the MMPU to address the physical memory. The maximum size of the physical address space is 2,097,152 bytes (2M) and it is addressed by a 20-bit address.

Address translation—The process of translating logical addresses into physical addresses.

Memory space—The addresses (physical or logical) assigned to a particular user.

Page—1024 (2000_8) words in memory.

User map—The set of memory address translation functions defined for a particular user.

Data channel map—The set of address translation functions defined for the memory references of a data channel used by a particular device.

Supervisor—The section of the operating system (software) which controls system functions such as the operation of the MMPU.

MMPU Functions

The MMPU's functions include address translation; the sharing of physical memory as a space-saving measure; the provision of user and data channel maps; operation in unmapped mode, primarily for diagnostic purposes; and system protection.

Address Translation

The primary function of the MMPU is address translation. The map divides each user's logical address space into 1024-word pages and correlates each logical page with a corresponding physical page. The address space the user sees is unchanged, but the map now translates each logical address into a physical address before memory is actually accessed.

Note that there is no requirement that the physical pages assigned to a user be in any particular order in physical memory. The supervisor can therefore use physical memory very flexibly, selecting unused pages for a new user without concern for maintaining any particular arrangement. Very complete use of the physical memory is also possible, since no contiguous blocks of memory larger than 1024 words are required.

Sharing of Physical Memory

The MMPU in the ECLIPSE S/140 is also capable of declaring a section of physical memory accessible to several users at once. This is useful if several users need a routine to perform some common function (e.g., trigonometric tables). Without this capability, each user would require a separate copy of the routine, thus creating many duplicate copies and wasting considerable space.

User and Data Channel Maps

Two types of maps are provided in the ECLIPSE S/140. *User maps* translate logical addresses to physical addresses when memory reference instructions are encountered in the user's program. *Data channel maps* translate logical addresses to physical addresses when data channel devices address the memory.

Each user requires a separate user map. The MMPU can hold two user maps, but only one can be enabled at any one time. Thus, if there are two users, the supervisor specifies a user map for each and loads it into the MMPU. The supervisor can then enable one or the other as needed. If there are more than two users, new user maps must be loaded as needed. In some operating systems, the operating system itself uses one of the user maps, so that a new user map must be loaded each time another user is serviced. This is not as much of an overhead burden as it sounds, because the *Load Map* instruction loads a complete map with one instruction, using relatively little time.

Separate data channel maps are needed because data channel devices can access memory without direct control from the user's program. There is thus no assurance that the proper user map would still be enabled at the time of the data channel request. The MMPU can hold four data channel maps. Enabling data channel mapping enables all four data channel maps at the same time. The choice of which map is used for data channel transfer is made by the I/O controller making the request. Those controllers not equipped to make this distinction use data channel map A by default. See the *Programmer's Reference Manual - Peripherals* (DGC No. 014-000632).

Operation in Unmapped Mode

So far we have assumed operation in the mapped mode. The MMPU can also operate in the unmapped mode. This mode is used for diagnostic purposes and for certain MMPU control functions. In unmapped mode, addresses in the range 0-75777_8 (which form logical pages 0-30) are not translated. In unmapped mode, addresses in the range 76000-77777_8 are translated by the special map for logical page 31. This allows you to access selected portions of user space while in unmapped mode.

System Protection

In addition to its address translation functions, the MMPU also provides protection functions. These generally protect the integrity of the system by preventing unauthorized access to certain parts of memory or to I/O devices. For example, if a set of trigonometric functions is stored in a section of memory accessible to all users, this section can be *write protected* so that users can read the functions, but cannot change them.

The various types of protection available in the ECLIPSE S/140 include validity, write, indirect, and I/O protection.

Validity protection protects a user's memory space from inadvertent access by another user, thereby preserving the integrity and privacy of the user's memory space. When a user's map is specified, the blocks of logical addresses required by the user's program are linked to blocks of physical addresses. The remaining (unused) logical blocks are declared invalid to that user, and an attempt to access them will cause a validity protection fault.

Validity protection is always enabled, so the supervisor's responsibility is limited to declaring the appropriate blocks of logical addresses invalid.

Write protection permits users to read the protected memory addresses, but not to write into them. In this way, the integrity of common areas of memory can be protected. An attempt to write into a write protected area of memory will cause a protection fault.

When the user map is loaded, its address space is automatically write protected. Write protection can be enabled or disabled by the supervisor.

Indirect protection prevents the CPU from becoming caught in an indirection loop that could stop instruction execution. An indirection loop occurs when the effective address calculation follows a chain of indirect addresses and never finds a word with bit 0 set to 0. Without indirect protection, the CPU would be unable to execute any further instructions, thus effectively halting the system until the console RESET switch is pressed.

With indirect protection enabled, a chain of 15 indirect references will cause a protection fault. Indirect protection can be enabled or disabled by the supervisor.

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. Clearly, it is undesirable to permit individual users to execute I/O instructions, since this will interfere with the operating system. If a user with I/O protection enabled attempts to execute an I/O instruction, a protection fault will occur. I/O protection can be enabled or disabled by the supervisor.

MMPU protection faults occur when a user attempts to violate one of the enabled types of protection. A protection fault takes the following form.

- The current user map is disabled.
- A 5-word return block is pushed onto the stack.
- Control is transferred to the protection fault handler, through an indirect jump via location 3.

The system programmer must supply the protection fault handler. It determines the type of fault that occurred (using the *Read Map Status* instruction), and then takes the appropriate action.

A protection fault can occur at any point during the execution of an instruction. Therefore, the return address in the fifth word of the return block is not always correct. For I/O protection faults, however, the fifth word will always be the logical address of the instruction following the instruction that caused the fault.

MMPU Instructions

The MMPU instructions control the actions of the MMPU. They are used by the supervisor program to change the mapping functions or check status of the various maps.

Load Effective Address Mode

The *Load Effective Address* **(LEF)** instruction has the same format as some of the I/O instructions. The MMPU therefore has a *Lef* mode bit which determines whether an I/O format instruction will be interpreted as an I/O or a **LEF** instruction. When the *Lef* mode bit is 1 (*Lef* mode enabled), all I/O format instructions are interpreted as *Load Effective Address* instructions. When the *Lef* mode bit is 0, all I/O format instructions are interpreted as I/O instructions.

The *Load Effective Address* instruction is very useful for quickly loading a constant into an accumulator. In addition, the *Lef* mode can be used for I/O protection. A user operating in the *Lef* mode is effectively denied access to any I/O devices, because all I/O and *Lef* instructions are interpreted as *Lef* instructions in this mode. Note, however, that no indication is given if an I/O instruction is interpreted as a *Lef* instruction.

When the MMPU is not operating in the *Lef* mode, all *Lef* and I/O instructions are interpreted as I/O instructions. With I/O protection enabled, these instructions will cause a protection fault in the normal manner. With I/O protection disabled, the *Lef* instruction will be executed as an I/O instruction, if possible.

Initial Conditions

At power up, the user maps and the data channel maps are undefined, the MMPU is in unmapped mode, and unmapped logical page 31 is mapped to physical page 31.

After an *I/O Reset*, the MMPU is in unmapped mode, the data channel maps are disabled, and unmapped logical page 31 is mapped to physical page 31.

NOTE: *MMPU instructions* can *be executed in mapped mode if I/O protection and* Lef *mode are disabled for that user. When executed in mapped mode, the* Read Map Status, Initiate Page Check, *and* Page Check *instructions will return the desired information without changing the map. The* Map Single Cycle *instruction will disable the user map after the next memory reference. The remainder of the instructions will change the map while the map is enabled, with undesirable results for this user, another user, or the system as a whole.*

Enabling Lef *mode only will convert all I/O instructions (including MMPU instructions) to* Lef *instructions. The* Load Map *instruction, however, does not use the I/O format and therefore can still be executed. Enabling both* Lef *mode and I/O protection will prevent execution of the* Load Map *instruction.*

The MMPU instructions are shown in Table 2.34. All except *Load Map* are in I/O format using the device mnemonic **MAP**.

Mnem	Instructions	Action
DIA	Read Map Status	Reads the status of the current map.
DIC	Page Check	Provides the identity and some characteristics of the physical page corresponding to the logical page identified by the immediately preceding *Initiate Page Check* instruction.
DOA	Load Map Status	Defines the parameters of a new map.
DOB	Map Supervisor Page 31	Specifies the physical page corresponding to logical page 31 of unmapped address space.
DOC	Initiate Page Check	Identifies a logical page; selects map without changing status.
LMP	Load Map	Loads successive words from memory into the MAP where they are used to define a user or data channel map.
NIOP	Map Single Cycle	Maps one memory reference using the last user map.

Table 2.34 MMPU instructions

Chapter 3
Optional Features

In this chapter we describe the optional facilities for the ECLIPSE S/140 and briefly discuss the instructions that program them. These options are:

- Floating Point Unit
- Character Instructions
- Burst Multiplexor Channel

Chapter 4 contains the descriptions, in dictionary form, of all the ECLIPSE S/140 instructions, except those that control I/O. Chapter 5 is the I/O instruction dictionary.

Floating Point Instructions

The floating point instruction set performs rapid arithmetic operations on numbers with a much larger range than the fixed point instruction set can handle. Single-precision floating point operations are capable of about 7 significant decimal digits, while double-precision operations are capable of about 16 significant decimal digits.

If the floating point instruction set is not installed, floating point instructions are executed as NO OPS.

We represent a floating point value using a 4-byte-wide (for single-precision) or an 8-byte-wide (for double-precision) number. The 4- or 8-byte aggregate contains three fields:

- A sign
- An exponent, which is adjusted to maintain the correct value of the number
- A fractional part called the mantissa, which, at the end of all floating point mathematical operations, is always adjusted to be greater than or equal to 1/16 and less than 1 (i.e., *normalized*)

Figure 3.1 shows these fields.

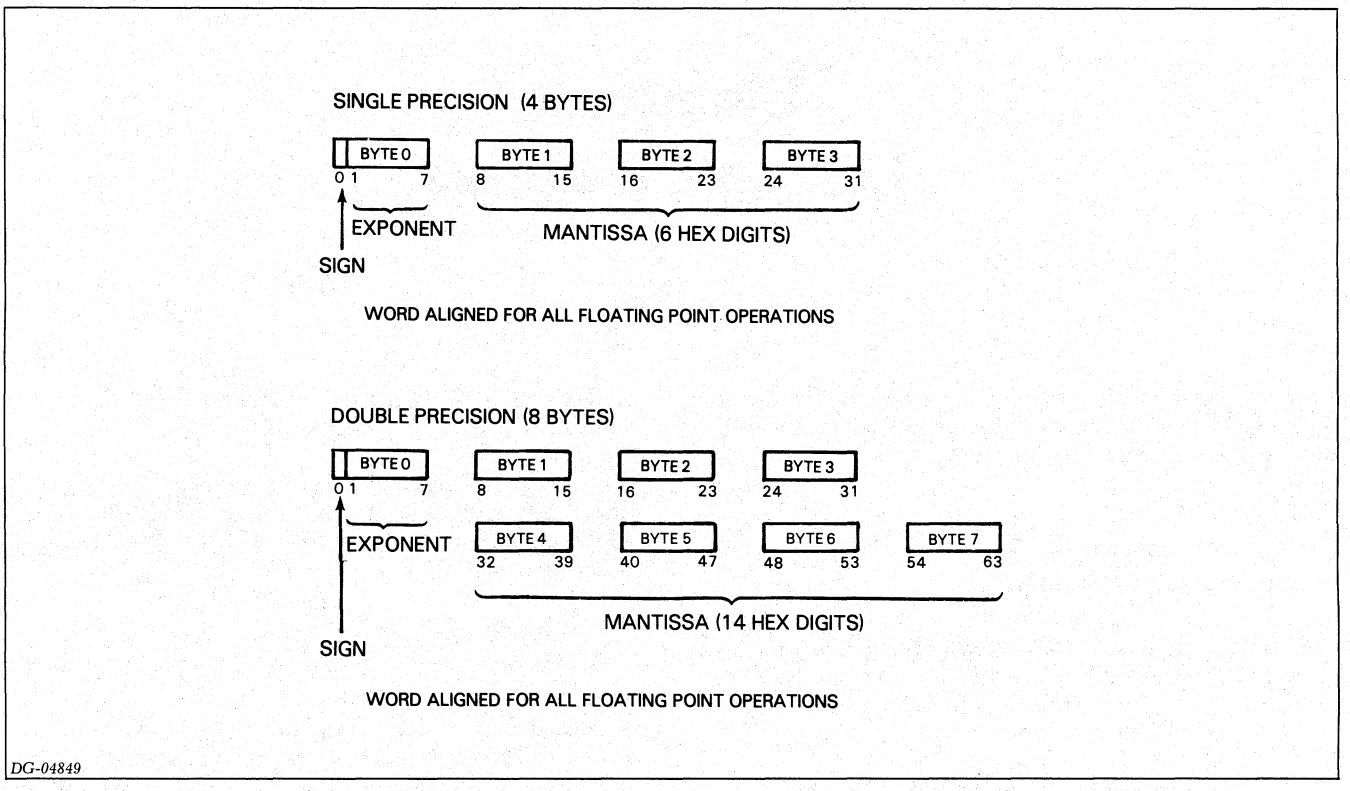


Figure 3.1 Floating point format

The magnitude of a floating point number is defined to be:

$$\text{Mantissa} \times 16^{(\text{True value of the exponent})}$$

We represent zero in floating point format by a number with all bits zero, known as *true zero*. When a calculation results in a zero mantissa, the number is automatically converted to a true zero.

The **sign** is bit 0 of the first byte. If the sign bit is 0, the number is positive. If the sign bit is 1, the number is negative.

The **exponent** is contained in the low-order seven bits of the first byte. We use *excess 64* representation. For both positive and negative exponents, the value is sixty-four greater than the true value of the exponent. Table 3.1 illustrates this.

Exponent Field	True Value of Exponent
0	-64
64	0
127	+63

Table 3.1 Exponent fields and values

The **mantissa** is contained in bytes 1-3 (single precision) or bytes 1-7 (double precision). By definition, the hexadecimal point lies *between* byte 0 and byte 1 of a floating point number.

To keep the mantissa in the range of 1/16 to 1, the results of each floating point calculation are *normalized*. A mantissa is normalized by shifting it left one hex digit (four bits) at a time, until the high-order four bits (the left-most four bits of byte 1) represent a nonzero quantity. For every hex digit shifted, the exponent is decreased by one.

Floating Point Arithmetic

Floating point arithmetic instructions require that the number be *word aligned*, so that bit 0 of the first byte of the number is bit 0 of the first word of a 2-word or 4-word area in memory.

NOTE: *The ECLIPSE S/140 floating point instructions assume normalized input numbers. Results are undefined if the input is not normalized.*

Floating Point Registers

There are five registers available to the programmer in the floating point processor. These are the four floating point accumulators (FPAC's) and the Floating Point Status Register (FPSR). The FPAC's are numbered 0-3 and are called FPAC0, FPAC1, FPAC2, and FPAC3. The FPSR is a 32-bit register that contains information about the present status of the floating point processor. Table 3.2 shows the FPSR.

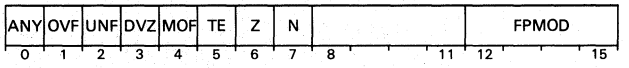
Guard Digit

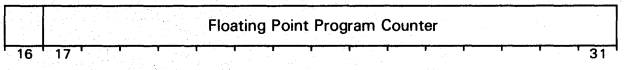
In order to increase accuracy, we use a 4-bit (one hex digit) *guard digit* during floating point arithmetic operations. The guard digit accepts and holds up to four bits shifted out (to the right) of the mantissa, and is used in all single precision and double precision operations. The processor truncates the guard digit before storing data at the end of the instruction.

Floating Point Fault Conditions

After every floating point operation, the floating point status register is checked for possible fault conditions. Four types of floating point fault conditions can be detected:

- Overflow
- Underflow
- Divide by zero
- Mantissa overflow





Bits	Name	Contents or Function
0	ANY	Indicates that any of bits 1-4 are set.
1	OVF	Overflow Indicator-while processing a floating point number, an exponent overflow occurred; the result is correct except the exponent is 128 too small.
2	UNF	Underflow Indicator-while processing a floating point number, an exponent underflow occurred; the result is correct except that the exponent is 128 too large.
3	DVZ	Divide by Zero-while processing a floating point number, a zero divisor was detected; division was aborted and the operands remain unchanged.
4	MOF	Mantissa Overflow-during a **FSCAL** instruction, a significant bit was shifted out of the high order end of the mantissa; this bit is also set during a *Fix* instruction if the result does not fit into the destination location.
5	TE	Trap Enable-If this bit is 1, setting any of bits 1-4 will result in a floating point fault.
6	Z	Zero bit-The result of the last floating point operation was zero.

Table 3.2 Floating point status register (see continuation)

Bits	Name	Contents or Function
7	N	Negative bit-The result of the last floating point operation was less than zero.
8-11*		Reserved for future use.
12-15	FPMOD	Indicates computer series supporting the floating point instruction set. 0000 S/200, C/300, S/230, C/330 0001 S/130, S/250 standard FP 0010 M/600, C/350, S/250 optional FP 0011 S/140 hardware FP 0100 Reserved for future use. 0101 8660 SP, 8661 SP 0110 C/150, S/250 standard EAU 0111 Reserved 1111 S/140 firmware FP
16		Reserved for future use.
17-31	FPPC	Floating Point Program Counter - This is the logical address of the last floating point instruction executed. In the event of a floating point fault, this is the address of the floating point instruction that caused the fault.

Table 3.2 Floating point status register (continued)

**These bits are used as internal flags by the floating point unit; preserve them when saving the state of the FPSR.*

Floating Point Trap

If the program has set bits 0 and 5 of the floating point status register to 1, any floating point fault condition initiates a floating point trap. When the fault occurs, the floating point unit saves the fault state until it detects the next floating point instruction that is *not* a Push Floating Point State (FPSH) or a Pop Floating Point State (FPOP). Then it pushes a return block onto the stack. Table 3.3 shows the format of the return block.

NOTE: *The return address is the address of the next floating point instruction to be executed. The address of the instruction that caused the fault is in the floating point status register.*

Word	Description
0	AC0
1	AC1
2	AC2
3	AC3
4	Bit 0: Carry; Bit 1-15: return address

Table 3.3 Return block format

NOTE: *When a floating point fault occurs and the trap enable bit is 1, the trap enable bit is set to 0 before control is transferred to the floating point error handler. The trap enable bit should be set to 1 before normal processing resumes.*

Next, the program jumps indirect via location 45_8. That location should contain the address of a software routine to handle the floating point fault. The fault handler remedies the fault condition and returns program control (via the return address in the return block) to the floating point instruction whose detection initiated the trap. That instruction is carried out.

Table 3.4 lists all of the floating point instructions. Several instructions have two forms: One ends in **S**, indicating single-precision and the other ends in **D**, indicating double-precision formats. Both instruction forms function identically.

Character Manipulation Instructions

Four character instructions manipulate strings of characters. That is, the instructions move and compare characters and words of arbitrary lengths. Each unique character in a string occupies one byte. These instructions:

- Compare one byte string to another.
- Move a byte string from one area of memory to another.
- Translate a character string from one data type to another.

Table 3.5 describes the four character instructions.

Mnem	Instructions	Action
FAB	Absolute Value	Sets the sign bit of an FPAC to 0.
FAMS FAMD	Add (memory to FPAC)	Adds the floating point number in memory to the floating point number in an FPAC.
FAS, FAD	Add (FPAC to FPAC)	Adds the floating point number in one FPAC to the floating point number in another FPAC.
FCLE	Clear Errors	Sets bits 0-4 of the FPSR TO 0.
FCMP	Compare Floating Point	Compares two floating point numbers and sets the Z and N flags accordingly.
FDMS FDMD	Divide (FPAC by memory)	Divides the floating point number in an FPAC by a floating point number in memory.
FDS FDD	Divide (FPAC by FPAC)	Divides the floating point number in one FPAC by the floating point number in another FPAC.
FEXP	Load Exponent	Places bits 1-7 of AC0 in bits 1-7 of the specified FPAC.
FFAS	Fix To AC	Converts the integer portion of a floating point number to a signed two's complement integer and places the result in an accumulator.
FFMD	Fix To Memory	Converts the integer portion of a floating point number to double precision integer format and stores the result in two memory locations.
FHLV	Halve	Divides the floating point number in FPAC by 2.
FINT	Integerize	Sets the fractional portion of the floating point number in the specified FPAC to zero and normalizes the result.
FLAS	Float From AC	Converts a signed two's complement number in an accumulator to a single precision floating point number.
FLDS FLDD	Load Floating Point	Copies a floating point number from memory to a specified FPAC.
FLMD	Float From Memory	Converts the contents of two memory locations in integer format to floating point format and places the result in a specified FPAC.
FLST	Load Floating Point Status	Copies the contents of two specified memory locations to the FPSR.
FMMS, FMMD	Multiply (FPAC by memory)	Multiplies the floating point number in FPAC by the floating point number in a memory.
FMOV	Move Floating Point	Moves the contents of one FPAC to another FPAC.
FMS, FMD	Multiply (FPAC by FPAC)	Multiplies the floating point number in one FPAC by the floating point number in another FPAC.
FNEG	Negate	Inverts the sign bit of the FPAC.
FNOM	Normalize	Normalizes the floating point number in FPAC.
FNS	No Skip	No operation.
FPOP	Pop Floating Point State	Pops an 18-word floating point block off the user stack and alters the state of the floating point unit.
FPSH	Push Floating Point State	Pushes an 18-word floating point block onto the user stack.
FRH	Read High Word	Places the high-order 16 bits of an FPAC in AC0.
FSA	Skip Always	Skips the next sequential word.

Table 3.4 Floating point instructions (see continuation)

Mnem	Instructions	Action
FSCAL	Scale	Shifts the mantissa of the floating point number in FPAC either right or left, depending upon the contents of bits 1-7 of AC0.
FSEQ	Skip On Zero	Skips the next sequential word if the Z flag of the FPSR is 1.
FSGE	Skip On Greater Than Or Equal To Zero	Skips the next sequential word if the N flag of the FPSR is 0.
FSGT	Skip On Greater Than Zero	Skips the next sequential word if both the Z and N flags of the FPSR are 0.
FSLE	Skip On Less Than Or Equal To Zero	Skips the next sequential word if either the Z flag or the N flag of the FPSR is 1.
FSLT	Skip On Less Than Zero	Skips the next sequential word if the N flag of the FPSR is 1.
FSMS, FSMD	Subtract (memory from FPAC)	Subtracts the floating point number in memory from the floating point number in an FPAC.
FSND	Skip On No Zero Divide	Skips the next sequential word if the divide by zero (DVZ) flag of the FPSR is 0.
FSNE	Skip On Non-Zero	Skips the next sequential word if the Z flag of the FPSR is 0.
FSNER	Skip On No Error	Skips the next sequential word if bits 1-4 of the FPSR are all 0.
FSNM	Skip On No Mantissa Overflow	Skips the next sequential word if the mantissa overflow (MOF) flag of the FPSR is 0.
FSNO	Skip On No Overflow	Skips the next sequential word if the overflow (OVF) flag of the FPSR is 0.
FSNOD	Skip On No Overflow And No Zero Divide	Skips the next sequential word if both the overflow (OVF) flag and the divide by zero (DVZ) flag of the FPSR are 0.
FSNU	Skip On No Underflow	Skips the next sequential word if the underflow (UNF) flag of the FPSR is 0.
FSNUD	Skip On No Underflow And No Zero Divide	Skips the next sequential word if both the underflow (UNF) flag and the divide by zero (DVZ) flag of the FPSR are 0.
FSNUO	Skip On No Underflow And No Overflow	Skips the next sequential word if both the underflow (UNF) flag and the overflow (OVF) flag of the FPSR are 0.
FSS, FSD	Subtract (FPAC from FPAC)	Subtracts the floating point number in one FPAC from the floating point number in another FPAC.
FSST	Store Floating Point Status	Copies the contents of the FPSR to two memory locations.
FSTS, FSTD	Store Floating Point	Copies the contents of a specified FPAC into memory.
FTD	Trap Disable	Sets the trap enable flag of the FPSR to 0.
FTE	Trap Enable	Sets the trap enable flag of the FPSR to 1.

Table 3.4 Floating point instructions (continued)

Mnem	Instructions	Action
CMP	Character Compare	Compare one string of characters in memory to another string.
CMT	Character Move Until True	Move a string of bytes from one area of memory to another until it finds a delimiter in a user-specified table or until the source string is exhausted.
CMV	Character Move	Move a string of bytes from one area of memory to another under control of the four accumulator values.
CTR	Character Translate	Translate a string of bytes from one data type to another; either move it to another area of memory or compare it to a second string of bytes.

Table 3.5 Character instructions

Burst Multiplexor Channel

The Burst Multiplexor Channel (BMC) is a high speed communications pathway which transfers data directly between main memory and high speed peripherals. It is controlled by the device controller performing the data transfer. No program control or CPU interaction is required except to set up the BMC's map tables. As a result, BMC data transfers are limited only by the memory speed. If the BMC and the CPU attempt to access memory at the same time, the CPU has priority, unless the BMC is in overdrive mode.

The BMC enters the overdrive mode when the number of concurrent service requests by the device controllers equals or exceeds the number selected by hardware switches on the BMC-ERCC board. This mode gives priority to the BMC. When the number of concurrent requests from device controllers drops below the figure selected for overdrive condition, the CPU regains priority.

The maximum data rate for the BMC is:

- Input: 200 nsec per word, or 5 Megawords/second
- Output: 100 nsec per word, or 10 Megawords/second

BMC Address Modes

The BMC has two address modes. In the unmapped (physical) mode, the BMC receives 20-bit addresses from the device controllers and passes them directly to memory. As each data word is transferred to or from memory, the BMC increments the destination address, causing successive words to move to or from consecutive locations in memory.

The other BMC address mode is mapped. When a controller initiates a data transfer, it can specify the mapped (logical) mode. The high order 10 bits of the logical address form a logical page number, which the BMC MAP translates into a 10-bit physical page number. This page number, combined with the 10 low order bits from the logical address, forms a 20-bit physical address which is passed to memory.

BMC MAP

The BMC contains its own MAP which consists of 16 map tables, each containing 32 map registers. It uses these map registers to translate logical page numbers into physical page numbers.

Each map register holds a 10-bit physical page number and a validity protection bit (the controlling program loads this information into the tables before I/O transfers begin). The BMC uses the logical page number as an index into the map table, and the contents of the selected map register become the 10 high-order bits of the physical address. If the device controller asks the BMC to access a map register that has its validity protection bit set, then the BMC will flag a validity protect error and terminate the transfer.

Note that when the BMC performs a mapped transfer, it increments the destination address after it moves each data word. If the increment causes an overflow out of the 10 low order bits, this selects a new map register for subsequent address translation. Depending on the contents of the map tables, this could mean that successive words are not transferred to or from consecutive pages in memory.

BMC Instructions

Map loads and dumps are initiated by an I/O Start command to the BMC. The BMC's Busy flag is set to 1 when a map load or dump is in progress. There is no Done flag and the BMC never causes program interrupts.

Device code 5 is assigned to the BMC. The assembler recognizes the mnemonic BMC for this device code.

The operation of the BMC is essentially transparent to a program executing in the host processor. The program must set up the map tables, i.e., load the map registers, but the operation of the BMC and its MAP are controlled by the device controller performing the data transfer. The table below summarizes the BMC instructions.

Mnem	Name	Function
DIC	Read Status	Places the BMC status in an accumulator.
DOA	Specify Low Order Address	Selects a map transfer operation and specifies the low order part of the memory address for loading or dumping the first map register.
DOB*	Specify High Order Address	Specifies the high order part of a memory address for loading or dumping the first map register.
DOB*	Specify Initial Map Register	Specifies the first map register of a group to be loaded or dumped.
DOC*	Specify Map Register Count	Specifies the number of map registers to be loaded or dumped.
DOC*	Load Status	Used for diagnostic purposes only.

Table 3.6 Burst multiplexor channel instructions

**These instructions are dependent on accumulator contents.*

Chapter 4
Standard Machine Instructions

This chapter lists all the standard instructions for the ECLIPSE S/140. They appear in alphabetical order according to the mnemonics recognized by the assembler. Chapter 5 contains all the I/O instructions.

For each instruction we include:

- The mnemonic recognized by the assembler.
- The bit format required.
- The format for any arguments involved.
- A functional description of each instruction.

Coding Aids

We use certain conventions and abbreviations throughout this chapter to help you properly code each instruction for Data General's assembler. Briefly, they are these:

Symbol	Means
[] or *[]*	Square brackets indicate that the enclosed symbol (e.g., *[,skip]*) is an optional operand or mnemonic. Code it only if you want to specify the option.
BOLD	Code operands or mnemonics printed in boldface exactly as shown. For example, code the mnemonic for the *Move* instruction: **MOV**.
italic	For each operand or mnemonic in italics, replace the item with a number or symbol that provides the assembler value you need for that item (e.g., the proper accumulator number, an address, etc.

We use the following abbreviations throughout this chapter:

Abbr	Meaning
i	Signed two's complement integer in the range -32,768 to 32,767; or unsigned in the range 0 to 65,535.
N	Integer in the range 0-3.
n	Integer in the range 1-4.
AC	Accumulator.
ACS	Source accumulator.
ACD	Destination accumulator.
FPAC	Floating point accumulator.
FACS	Floating point source accumulator
FACD	Floating point destination accumulator.

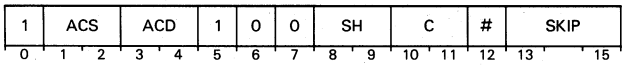
Setting the Index Field

To set the index field, code a comma followed by an integer between 0-3. This will set the index field to the value you specified. You can also use the symbol *dot* (.) to set the index field to 01 (PC relative). *Dot* can be read as the *address of the current instruction*. When you use *dot*, you usually follow it with a plus or minus sign and the displacement value, such as **.+3** or **.-12**.

If you are coding extended class instructions, note that using a dot (e.g., **EJMP .+5**) does not produce the same effect as coding a comma followed by a 1 (**EJMP 5,1**). When using a dot, the displacement is added to the address of the instruction (the first word of a two-word instruction). When using a comma, the displacement is added to the address of the word containing the displacement (the second word of a two-word instruction). Therefore, **EJMP .+5** is equivalent to **EJMP 4,1**.

Add Complement

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



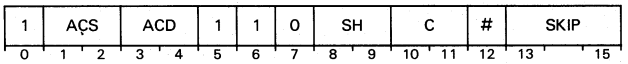
Adds the logical complement of an unsigned integer to another unsigned integer.

Initializes carry to the specified value, adds the logical complement of the unsigned, 16-bit number in ACS to the unsigned, 16-bit number in ACD, and places the result in the shifter. If the addition produces a carry of 1 out of the high-order bit, carry is complemented. 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.*

Add

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



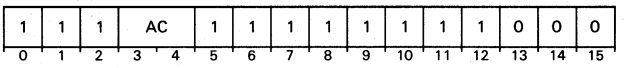
Performs unsigned integer addition, and complements carry if appropriate.

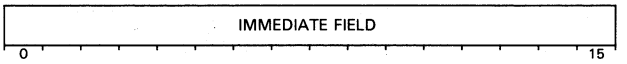
Initializes carry to the specified value, adds the unsigned, 16-bit number in ACS to the unsigned, 16-bit number in ACD, and places the result in the shifter. If the addition produces a carry of 1 out of the high-order bit, carry is complemented. The instruction then performs the specified shift operation 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.*

Extended Add Immediate

ADDI *i,ac*



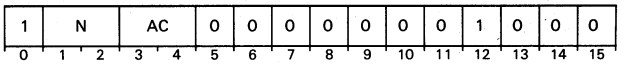


Adds a signed integer in the range -32,768 to +32,767 to the contents of an accumulator.

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.

Add Immediate

ADI *n,ac*



Adds an unsigned integer in the range 1-4 to the contents of an accumulator.

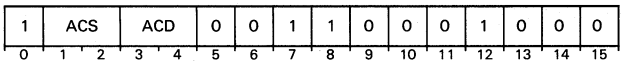
Adds the contents of the immediate field *N*, plus 1, to the unsigned, 16-bit number contained in the specified accumulator, placing the result in the same accumulator. Carry remains unchanged.

NOTE: *The assembler takes the coded value of* **n** *and subtracts one from it before placing it in the immediate field. Therefore, you should code the exact value that you wish to add.*

Example - Assume that AC2 contains 177775_8. After the instruction **ADI 4,2** is executed, AC2 contains 000001_8 and carry is unchanged.

AND With Complemented Source

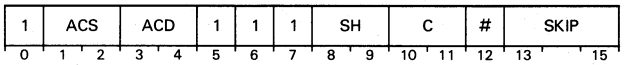
ANC *acs,acd*



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 bit position in ACS contains 0. The contents of ACS remain unchanged.

AND

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

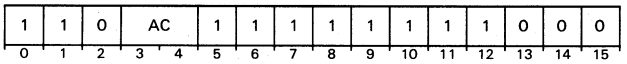


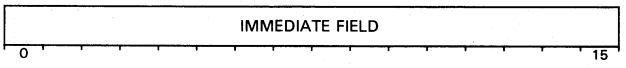
Forms the logical AND of the contents of two accumulators.

Initializes the carry bit to the specified value and places the logical AND of ACS and ACD in the shifter. Each bit placed in the shifter is 1 only if the corresponding bit in both ACS and ACD is 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.

AND Immediate

ANDI *i,ac*

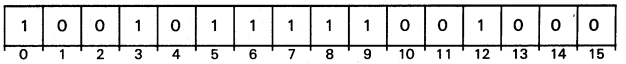




Places the logical AND of the contents of the immediate field and the contents of the specified accumulator in the specified accumulator.

Block Add and Move

BAM



Moves memory words from one location to another, adding a constant to each one.

Moves words sequentially from one memory location to another, treating them as unsigned, 16-bit integers. After fetching a word from the source location, the instruction adds the unsigned, 16-bit integer in AC0 to it. If the addition produces a carry of 1 out of the high-order bit, 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. In that case, 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.

AC	Contents
0	Addend
1	Number of words to be moved
2	Source address
3	Destination address

For each word moved, the count in AC1 is decremented by one and the source and destination addresses in AC2 and AC3 are incremented by one. Upon completion of the instruction, AC1 contains zeroes, and AC2 and AC3 point to the word following the last word in their respective fields. The contents of 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 may overlap in any way.

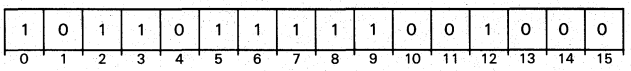
NOTE: *This instruction may require a long execution time. Another process can therefore interrupt it. If a* Block Add and Move *instruction is interrupted, the program counter is decremented by one before it is placed in location 0, so that it points to the interrupted instruction. Because the addresses and the word count*

are updated after every word stored, any interrupt service routine that returns control to the interrupted program via the address stored in memory location 0 will correctly restart the **BAM** *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 instruction will not try to resolve an indirect address in either AC2 or AC3.*

Block Move

BLM



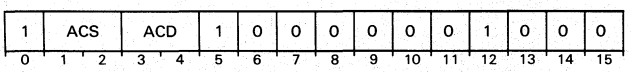
Moves memory words from one location to another.

The *Block Move* instruction is the same as the *Block Add And Move* instruction in all respects except that no addition is performed and AC0 is not used.

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

Set Bit To One

BTO *acs,acd*



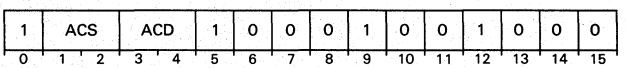
Sets the specified bit to 1.

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

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

Set Bit To Zero

BTZ *acs,acd*



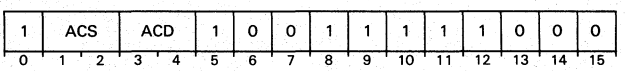
Sets the addressed bit to 0.

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

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

Compare To Limits

CLM *acs,acd*



Compares a signed integer with two other integers and skips if the first integer is between the other two. The accumulators determine the location of the three integers.

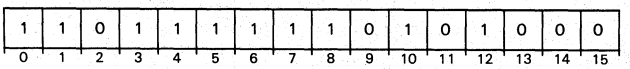
Compares the signed, two's complement integer in ACS to two signed, two's complement limit values, *L* and *H*. If the number in ACS is greater than or equal to *L* and less than or equal to *H*, the next sequential word is skipped. If the number in ACS is less than *L* or greater than *H*, the next 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. The limit value *H* is contained in the word following *L*. Bit 0 of ACD is ignored.

If ACS and ACD are specified as the same accumulator, then the integer to be compared must be in that AC, 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.

Character Compare

CMP



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

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

The four accumulators contain parameters passed to the instruction. Two accumulators specify the starting address, the number of bytes, and the direction of processing (ascending or descending addresses) for each string. Carry is used as an indicator.

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 instruction uses carry as an indicator.

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

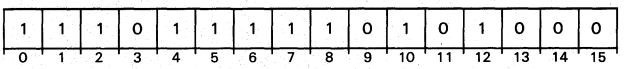
The strings may overlap in any way. Overlap will not effect the results of the comparison.

Upon completion, AC0 contains the number of bytes left to compare in string 2. AC1 contains the return code as shown in the table above. AC2 contains a byte pointer either to the failing byte in string 2 (if an inequality is found), or to the byte following string 2 (if string 2 is exhausted). AC3 contains a byte pointer either to the failing byte in string 1 (if an inequality is found), or to the byte following string 1 (if string 1 is exhausted). Carry contains an indeterminate value.

If the lengths of both strings 1 and 2 are zero, the instruction 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.

Character Move Until True

CMT



Under control of the four accumulators, moves a string of bytes from one area of memory to another until either a table-specified delimiter character is moved 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 $0\text{-}255_{10}$) 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 contain parameters passed to the instruction.

AC0 contains the address (word address), possibly indirect, of the start of the 256-bit (16-word) delimiter table.

AC1 specifies the length of the strings and the direction of processing. If the source string is to be moved to the destination string 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 string in descending order, AC1 contains the two's complement of the number of bytes in the source string.

AC2 contains 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 string. When the process is performed in descending order, AC2 points to the highest byte in the destination string.

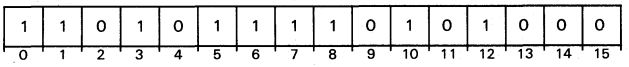
AC3 contains a byte pointer to the first byte to be processed in the source string. When the process is performed in ascending order, AC3 points to the lowest byte in the source string. When the process is performed in descending order, AC3 points to the highest byte in the source string.

If the strings overlap in any way, a trap occurs. When the source and destination addresses are the same (i.e., AC2=AC3), no data is moved. Any other type of overlap may produce unusual side effects.

Upon completion, AC0 contains the resolved address of the translation table and AC1 contains the number of bytes that were not moved. AC2 contains a byte pointer to the byte following the last byte written in the destination string. AC3 contains a byte pointer either to the delimiter or to the first byte following the source string.

Character Move

CMV



Under control of the four accumulators, moves a string of bytes from one area of memory to another and returns a value in the Carry bit reflecting the relative lengths of source and destination strings.

The instruction copies the source string to the destination field, one byte at a time. The four accumulators contain parameters passed to the instruction. Two accumulators specify the starting address, number of bytes to be copied, and the direction of processing (ascending or descending addresses) for each field.

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

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

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

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

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

Upon completion, AC0 contains 0 and AC1 contains the number of bytes left to fetch from the source field. AC2 contains a byte pointer to the byte following the destination field; and AC3 contains a byte pointer to the byte following the last byte fetched from the source field.

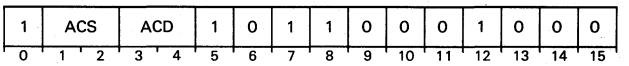
> **NOTE:** *If AC0 contains the number 0 at the beginning of this instruction, no bytes are fetched and none are stored. If AC1 is 0 at the beginning of this instruction, the destination field is filled with space characters.*

If the source field is longer than the destination field, the instruction terminates when the destination field is filled and sets carry to 1. In any other case, the instruction sets carry to 0.

If the source field is shorter than the destination field, the instruction pads the destination field with space characters $<040_8>$.

Count Bits

COB *acs,acd*

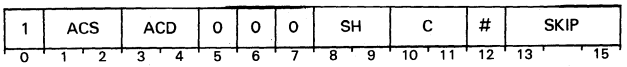


Adds a number equal to the number of ones in ACS to the signed, 16-bit, two's complement number in ACD. The instruction leaves the contents of ACS and the state of carry unchanged.

> **NOTE:** *If ACS and ACD are the same accumulator, the instruction functions as described above, except the contents of ACS will be changed.*

Complement

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

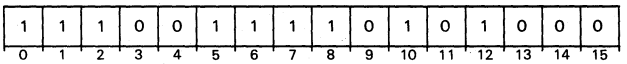


Forms the logical complement of the contents of an accumulator.

Initializes carry to the specified value, forms the logical complement of the number in 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.

Character Translate

CTR



Under control of the four accumulators, translates a string of bytes from one data type to another and either moves it to another area of memory or compares it to a second translated string.

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

When operating in translate and move mode, the instruction translates each byte in string 1, and places it in a corresponding position in string 2. Translation is performed by using each byte as an 8-bit index into a 256-byte translation table. The byte addressed by the index then becomes the translated value.

When operating in translate and compare mode, the instruction translates each byte in string 1 and string 2 as described above, and compares the translated values. Each translated byte is treated as an unsigned 8-bit binary quantity in the $0\text{-}255_{10}$. 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.

ACO specifies the address, either direct or indirect, of a word which contains a byte pointer to the first byte in the 256-byte translation table.

AC1 specifies the length of the two strings and the mode of processing. If string 1 is to be processed in translate and move mode, AC1 contains the two's complement of the number of bytes in the strings. If the strings are to be processed in translate and compare mode, AC1 contains the unsigned value of the number of bytes in the strings. Both strings are processed from lowest memory address to highest.

AC2 contains a 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, AC0 contains the address of the word which contains the byte pointer to the translation table and AC1 contains 0. AC2 contains a byte pointer to the byte following string 2 and AC3 contains a byte pointer to the byte following string 1.

Upon completion of a translate and compare operation, AC0 contains the address of the word which contains the byte pointer to the translation table. AC1 contains a return code as calculated in the table below. AC2 contains a byte pointer to either the failing byte in string 2 (if an inequality was found) or the byte following string 2 if the strings were identical. AC3 contains a byte pointer to either the failing byte in string 1 (if an inequality was found) or the byte following string 1 if the strings were identical.

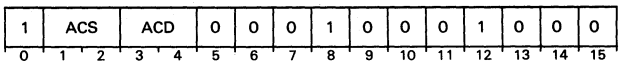
Code	Result
-1	Translated value of string 1 is less than the translated value of string 2.
0	Translated value of string 1 is equal to the translated value of string 2.
+1	Translated value of string 1 is greater than the translated value of string 2.

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

The fields may overlap in any way. However, processing is done one character at a time, so unusual side effects may be produced by certain types of overlap.

Decimal Add

DAD *acs,acd*



Performs decimal addition on 4-bit binary coded decimal (BCD) numbers and uses the carry bit for a decimal carry.

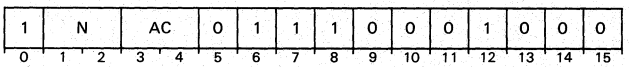
Adds the unsigned decimal digit contained in ACS bits 12-15 to the unsigned decimal digit contained in ACD bits 12-15. The carry bit is added to this result. The instruction places the decimal unit result in ACD bits 12-15, and the decimal carry in the carry bit. The contents of ACS and bits 0-11 of 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 — Assume that bits 12-15 of AC2 contain 9; bits 12-15 of AC3 contain 7; and the carry bit is 0. After the instruction **DAD 2,3** is executed, AC2 remains the same; bits 12-15 of AC3 contain 6; and the carry bit is 1, indicating a decimal carry from this *Decimal Add*.

Double Hex Shift Left

DHXL *n,ac*



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 zeroes.

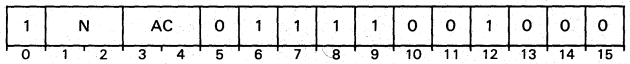
NOTE: *If AC is specified as AC3, then AC+1 is AC0.*

The assembler takes the coded value of n *and subtracts one from it before placing it in the immediate field. Therefore, the programmer should code the exact number of hex digits that he wishes to shift.*

If n *is equal to 3, the contents of AC+1 are placed in AC and AC+1 is filled with zeroes.*

Double Hex Shift Right

DHXR *n,ac*



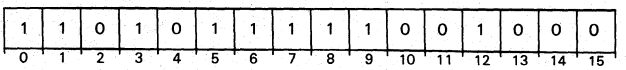
Shifts the 32-bit number contained in AC and AC+1 right a number of hex digits depending upon the immediate field *N*. The number of digits shifted is equal to *N+1*. Bits shifted out are lost and the vacated bit positions are filled with zeroes.

NOTE: *If AC is specified as AC3, then AC+1 is AC0. The assembler takes the coded value of* n *and subtracts one from it before placing it in the immediate field. Therefore, the programmer should code the exact number of hex digits that he wishes to shift.*

If N *is equal to 3, the contents of AC are placed in AC+1 and AC is filled with zeroes.*

Unsigned Divide

DIV



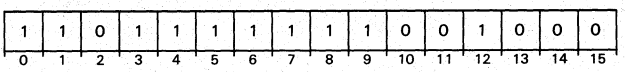
Divides the unsigned 32-bit integer in two accumulators by the unsigned contents of a third accumulator. The quotient and remainder each occupy one accumulator.

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 to 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.*

Signed Divide

DIVS



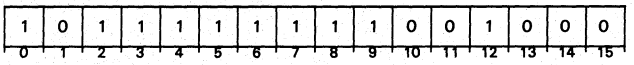
Divides the signed 32-bit integer in two accumulators by the signed contents of a third accumulator. The quotient and remainder each occupy one accumulator.

The signed, 32-bit two's complement number contained in AC0 and AC1 is divided by the signed, 16-bit two's complement number 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, except that a zero quotient or a zero 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.*

Sign Extend and Divide

DIVX

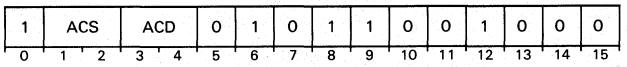


Extends the sign of one accumulator into a second accumulator and performs a *Signed Divide* on the result.

Extends the sign of the 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 *Signed Divide* operation.

Double Logical Shift

DLSH *acs,acd*



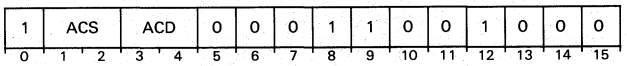
Shifts the 32-bit number contained in ACD and ACD+1 either left or right depending on the number contained in bits 8-15 of ACS. The signed, 8-bit two's complement number contained in bits 8-15 of ACS determines the direction of the shift and the number of bits to be shifted. If the number in bits 8-15 of ACS is positive, shifting is to the left; if the number in bits 8-15 of ACS is negative, shifting 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 zeroes. Carry and 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 the contents of ACS remain unchanged.*

Decimal Subtract

DSB *acs,acd*



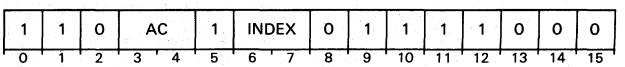
Performs decimal subtraction on 4-bit binary coded decimal (BCD) numbers and uses carry as a decimal borrow.

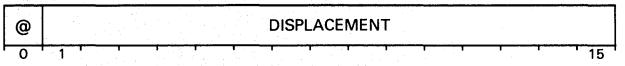
Subtracts the unsigned decimal digit contained in ACS bits 12-15 from the unsigned decimal digit contained in ACD bits 12-15. Subtracts the complement of carry from this result. Places the decimal unit position of the final result in ACD bits 12-15 and the complement of the decimal borrow in carry. In 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. In addition, the result is in ten's complement form (i.e., it is ten greater than the actual binary result.)

Example — Assume that bits 12-15 of AC2 contain 9; bits 12-15 of AC3 contain 7; and carry contains 0. After the instruction **DSB 3,2** is executed, AC3 remains the same; bits 12-15 of AC2 contain 1; and carry is set to 1, indicating no borrow from this *Decimal Subtract*.

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*. The dispatch table consists of a table of addresses. Immediately before the table are two signed, two's complement limit words, *L* and *H*. The last word of the table is in location *E+H-L*, as shown in Figure 4.1.

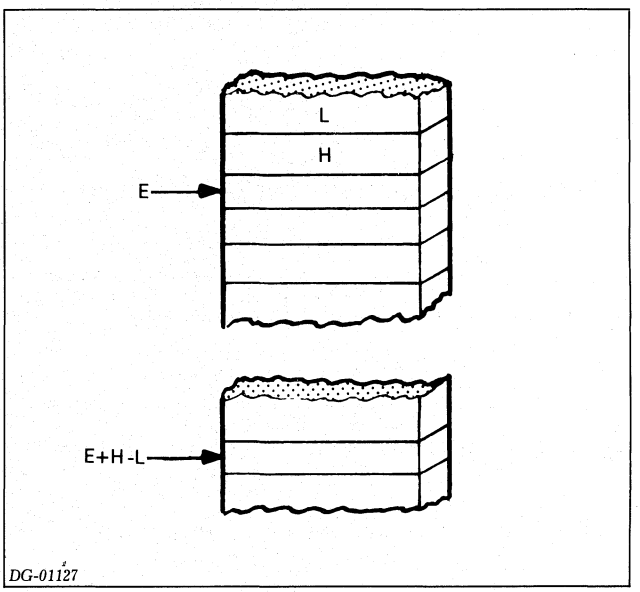


Figure 4.1 Diagram of dispatch table

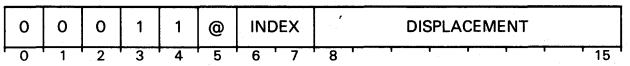
Compares the signed, two's complement number contained in the accumulator to the limit words. If the number in the accumulator is less than *L* or greater than *H*, sequential operation continues with the instruction immediately after the *Dispatch* instruction.

If the number in AC 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.

Decrement And Skip If Zero

DSZ *[@]displacement[,index]*

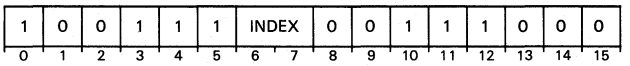


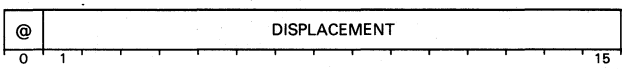
Decrements the addressed word, then skips if the decremented value is zero.

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

Extended Decrement and Skip if Zero

EDSZ *[@]displacement[,index]*



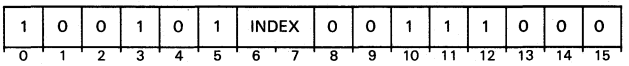


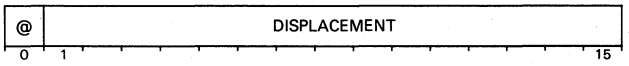
Decrements the addressed word, then skips if the decremented value is zero.

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

Extended Increment And Skip If Zero

EISZ *[@]displacement[,index]*



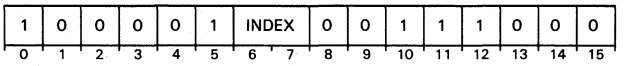


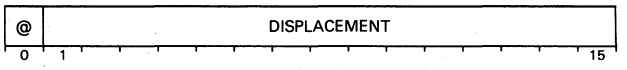
Increments the addressed word, then skips if the incremented value is zero.

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

Extended Jump

EJMP *[@]displacement[,index]*

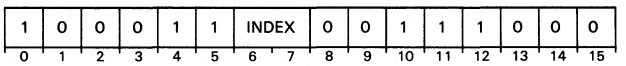


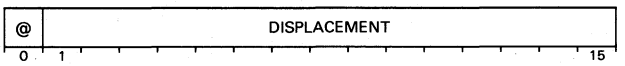


Computes the effective address, *E*, and places it in the program counter. Sequential operation continues with the word addressed by the updated value of the program counter.

Extended Jump To Subroutine

EJSR *[@]displacement[,index]*





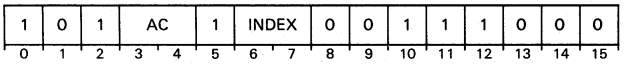
Increments and stores the value of the program counter in AC3, then places a new address in the program counter.

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

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

Extended Load Accumulator

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

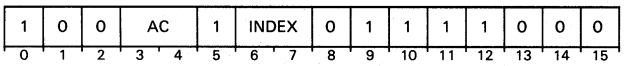


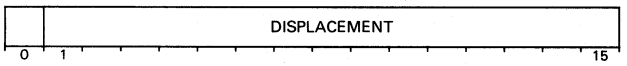
Moves a copy of the contents of a memory word into the specified accumulator.

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.

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.

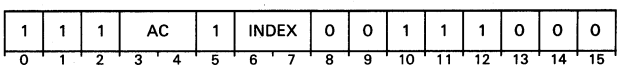
The instruction destroys the previous contents of the specified accumulator, but it does not alter either the index value or the displacement.

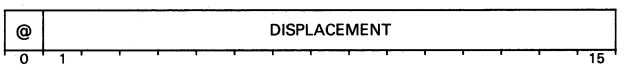
The argument *index* selects the source of the index value. It may have values in the range of 0-3. The meaning of each value is shown below:

Index Bits	Index Value
00	0
01	Address of the displacement field (Word 2 of this instruction)
10	Contents of AC2
11	Contents of AC3

Extended Load Effective Address

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





Places an effective address in an accumulator.

Computes the effective address, *E*, and places it in bits 1-15 of the specified accumulator. Sets bit 0 of the accumulator to 0. The previous contents of the accumulator are lost.

Figure 4.2 shows some different uses of the **ELEF** instruction.

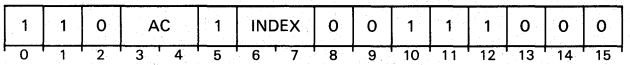
```
ELEF    0,TABLE     ;The logical address of TABLE
                    ;is placed in AC0.
ELEF    1,-55,3     ;Subtracts 000055 (octal) from
                    ;the unsigned integer in AC3 and
                    ;places the result in AC1.
ELEF    0,.+0       ;Places the logical address of this
                    ;Load effective address
                    ;instruction in AC0.
```

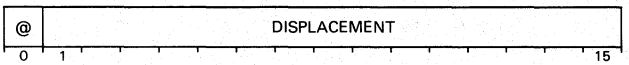
DG-06562

Figure 4.2 Examples of ELEF instruction

Extended Store Accumulator

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



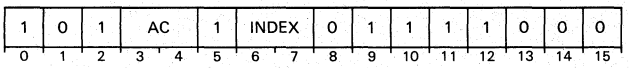


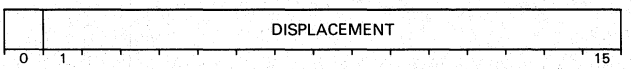
Stores the contents of an accumulator into a memory location.

The contents of the specified accumulator are placed in the word addressed by the effective address, *E*. The previous contents of the location addressed by *E* are lost. The contents of the specified accumulator remain unchanged.

Extended Store Byte

ESTB *ac,displacement[,index]*





Copies into memory the byte contained in the right half of an accumulator.

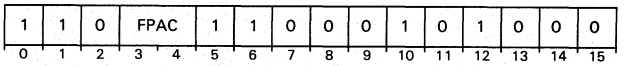
Forms a byte pointer from the displacement as follows: shifts the 16-bit number contained in the displacement field to the right one bit, producing a 15-bit address and a 1-bit byte indicator. Uses the value of the index bits to determine an offset value. Adds the offset value to the 15-bit address produced from the displacement field to give a memory address. The byte indicator determines which byte of the addressed location will receive bits 8-15 of the specified accumulator.

The argument *index* selects the source of the index value. It may have values in the range of 0-3; the meaning of each value is as follows:

Index Bits	Index Value
00	0
01	Address of the displacement field (Word 2 of this instruction)
10	Contents of AC2
11	Contents of AC3

Absolute Value

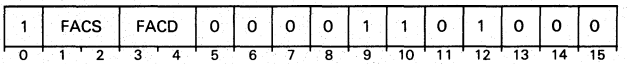
FAB *fpac*



Sets the sign bit of FPAC to 0. Also sets the exponent to zero if the mantissa is zero; otherwise leaves bits 1-63 of FPAC unchanged. Updates the **Z** and **N** flags in the floating point status register to reflect the new contents of FPAC.

Add Double (FPAC to FPAC)

FAD *facs,facd*



Adds the floating point number in FACS to the floating point number in FACD and places the normalized result in FACD. Destroys the previous contents of FACD, leaves the contents of FACS unchanged and updates the **Z** and **N** flags in the floating point status register to reflect the new contents of FACD.

Floating point addition consists of an exponent comparison and a mantissa addition. The exponents of the two numbers are compared, and the mantissa of the number with the smaller exponent is shifted right. This mantissa alignment is accomplished by taking the absolute value of the difference between the two exponents and shifting the mantissa right that number of hex digits. One guard digit is provided so that all but four bits shifted out of the right end of the mantissa are lost, and do not take part in the addition. If all significant digits are shifted out of the mantissa, the operation is equivalent to adding the number with the larger exponent to zero. This requires a shift of at least 15 hex digits.

After alignment, the mantissas are added together. The result of this addition is termed the intermediate result. One guard digit is provided for the intermediate result, which is used if normalization is required. The sign of the intermediate result is determined from the signs of the two operands by the rules of algebra. If the mantissa addition produces a carry out of the high-order bit, the mantissa in the intermediate result is shifted right one hex

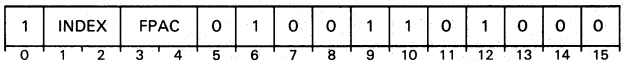
digit and the exponent is incremented by one. If this shift produces an exponent overflow, the **OVF** bit is set in the floating point status register, and the number in FACD is correct, except that the exponent is 128 too small.

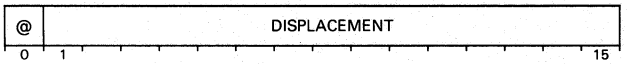
If there is no mantissa overflow, the mantissa of the intermediate result is examined for leading hex zeros. If the mantissa is found to be all zeros, a true zero is placed in the FACD and the instruction terminates.

If the mantissa is non-zero, the intermediate result is normalized, and the number placed in the FACD. If the normalization results in an exponent underflow, the **UNF** bit is set in the floating point status register and the instruction is terminated. The number in the FACD is correct except that the exponent is 128 too large.

Add Double (Memory to FPAC)

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





Adds the floating point number in the source location to the floating point number in FPAC and places the normalized result in FPAC. Destroys the previous contents of FPAC, leaves the contents of the source location unchanged and updates the **Z** and **N** flags in the floating point status register to reflect the new contents of FPAC.

Computes the effective address *E* which addresses a 4-word (double precision) operand.

Floating point addition consists of an exponent comparison and a mantissa addition. The exponents of the two numbers are compared, and the mantissa of the number with the smaller exponent is shifted right. This mantissa alignment is accomplished by taking the absolute value of the difference between the two exponents and shifting the mantissa right that number of hex digits. One guard digit is provided so that all but four bits shifted out of the right end of the mantissa are lost, and do not take part in the addition. If all significant digits are shifted out of the mantissa, the operation is equivalent to adding the number with the larger exponent to zero. This requires a shift of at least 15 hex digits.

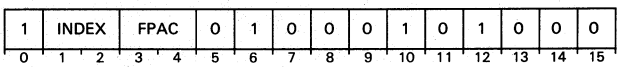
After alignment, the mantissas are added together. The result of this addition is termed the intermediate result. One guard digit is provided for the intermediate result, which is used if normalization is required. The sign of the intermediate result is determined from the signs of the two operands by the rules of algebra. If the mantissa addition produces a carry out of the high-order bit, the mantissa in the intermediate result is shifted right one hex digit and the exponent is incremented by one. If this shift produces an exponent overflow, the **OVF** bit is set in the floating point status register, and the number in FPAC is correct except that the exponent is 128 too small.

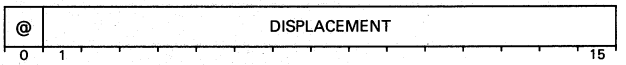
If there is no mantissa overflow, the mantissa of the intermediate result is examined for leading hex zeros. If the mantissa is found to be all zeros, a true zero is placed in the FPAC and the instruction terminates.

If the mantissa is non-zero, the intermediate result is normalized, and the number placed in the FPAC. If the normalization results in an exponent underflow, the **UNF** bit is set in the floating point status register and the instruction is terminated. The number in the FPAC is correct except that the exponent is 128 too large.

Add Single (Memory to FPAC)

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





Adds the floating point number in the source location to the floating point number in FPAC and places the normalized result in FPAC. Destroys the previous contents of FPAC, leaves the contents of the source location unchanged and updates the **Z** and **N** flags in the floating point status register to reflect the new contents of FPAC.

Computes the effective address, *E*, which addresses a 2-word (single precision) operand.

Floating point addition consists of an exponent comparison and a mantissa addition. The exponents of the two numbers are compared, and the mantissa of the number with the smaller exponent is shifted right. This mantissa alignment is accomplished by taking the absolute value of the difference between the two exponents and shifting the mantissa right that number of hex digits. One guard digit is provided so that all but four bits shifted out of the right end of the mantissa are lost, and do not take part in the addition.

If all significant digits are shifted out of the mantissa, the operation is equivalent to adding the number with the larger exponent to zero. This requires a shift of 7 hex digits.

After alignment, the mantissas are added together. The result of this addition is termed the intermediate result. One guard digit is provided for the intermediate result, which is used if normalization is required. The sign of the

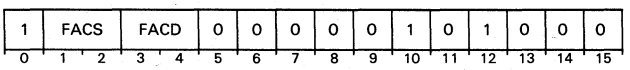
intermediate result is determined from the signs of the two operands by the rules of algebra. If the mantissa addition produces a carry out of the high-order bit, the mantissa in the intermediate result is shifted right one hex digit and the exponent is incremented by one. If this shift produces an exponent overflow, the **OVF** bit is set in the floating point status register, and the number in FPAC is correct, except that the exponent is 128 too small.

If there is no mantissa overflow, the mantissa of the intermediate result is examined for leading hex zeros. If the mantissa is found to be all zeros, a true zero is placed in the FPAC and the instruction terminates.

If the mantissa is non-zero, the intermediate result is normalized, and the number placed in the FPAC. If the normalization results in an exponent underflow, the **UNF** bit is set in the floating point status register and the instruction is terminated. The number in the FPAC is correct except that the exponent is 128 too large.

Add Single (FPAC to FPAC)

FAS *facs,facd*



Adds the floating point number in FACS to the floating point number in FACD and places the normalized result in FACD. Destroys the previous contents of FACD, leaves the contents of FACS unchanged and updates the **Z** and **N** flags in the floating point status register to reflect the new contents of FACD.

Floating point addition consists of an exponent comparison and a mantissa addition. The exponents of the two numbers are compared, and the mantissa of the number with the smaller exponent is shifted right. This mantissa alignment is accomplished by taking the absolute value of the difference between the two exponents and shifting the mantissa right that number of hex digits. One guard digit is provided so that all but four bits shifted out of the right end of the mantissa are lost, and do not take part in the addition. If all significant digits are shifted out of the mantissa, the operation is equivalent to adding the number with the larger exponent to zero. This requires a shift of 7 hex digits.

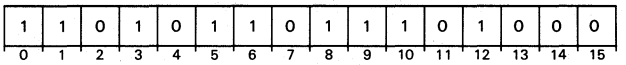
After alignment, the mantissas are added together. The result of this addition is termed the intermediate result. One guard digit is provided for the intermediate result, which is used if normalization is required. The sign of the intermediate result is determined from the signs of the two operands by the rules of algebra. If the mantissa addition produces a carry out of the high-order bit, the mantissa in the intermediate result is shifted right one hex digit and the exponent is incremented by one. If this shift produces an exponent overflow, the **OVF** bit is set in the floating point status register, and the number in FACD is correct, except that the exponent is 128 too small.

If there is no mantissa overflow, the mantissa of the intermediate result is examined for leading hex zeros. If the mantissa is found to be all zeros, a true zero is placed in the FACD and the instruction is terminated.

If the mantissa is non-zero, the intermediate result is normalized, and the number placed in the FACD. If the normalization results in an exponent underflow, the **UNF** bit is set in the floating point status register and the instruction is terminated. The number in the FACD is correct, except that the exponent is 128 too large.

Clear Errors

FCLE

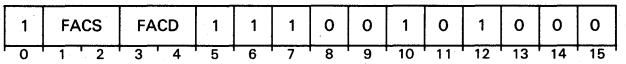


Sets bits 0-4 of the floating point status register to 0.

NOTE: *The I/O RESET instruction will also set these bits to 0.*

Compare Floating Point

FCMP *facs,facd*



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

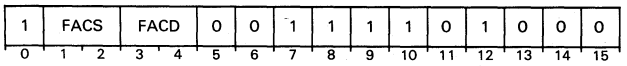
Algebraically compares the floating point numbers in FACS and FACD to each other and updates the **Z** and **N** flags in the floating point status register to reflect the result. Leaves the contents of FACS and FACD unchanged. The results of the compare and the corresponding flag settings are shown below.

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

NOTE: *Unnormalized operands give unspecified results.*

Divide Double (FPAC by FPAC)

FDD *facs,facd*



Divides the floating point number in FACD by the floating point number in FACS and places the normalized result in FACD. Destroys the previous contents of FACD, leaves the contents of FACS unchanged, and updates the **Z** and **N** flags in the floating point status register to reflect the new contents of FACD.

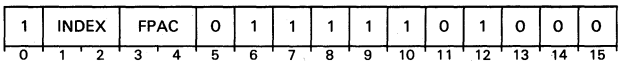
The source operand is checked for a zero mantissa. If the mantissa is zero, the **DVZ** bit is set in the floating point status register and the instruction is terminated. The number in FACD remains unchanged. If the mantissa is nonzero, the previous contents of FACD are lost. The two mantissas are compared and if the mantissa of the number in FACD is greater than or equal to the mantissa of the source operand, the mantissa of the number in FACD is shifted right one hex digit and the exponent of the number in FACD is increased by one.

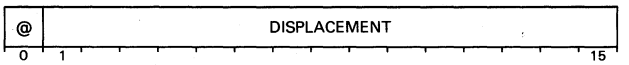
The mantissa in FACD is then divided by the mantissa of the source operand and the quotient is the mantissa of the intermediate result. The exponent of the source operand is subtracted from the exponent in FACD and 64 is added to this result. This addition of 64 maintains the *excess 64* notation. The result of the exponent manipulation becomes the exponent of the intermediate result. The sign of the intermediate result is determined from the signs of the two operands by the rules of algebra. The result is placed in FACD. (Because the operands are assumed to be normalized, and division with such operands produces a normalized result, no normalization of the result takes place.)

If the exponent processing produces either overflow or underflow, the corresponding bit in the floating point status register is set. The number in FACD is correct except that, for exponent overflow, the exponent is 128 too small, and for exponent underflow, the exponent is 128 too large.

Divide Double (FPAC by Memory)

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





Divides the floating point number in FPAC by the floating point number in the source location and places the normalized result in FPAC. Destroys the previous contents of FPAC, leaves the contents of the source location unchanged, and updates the **Z** and **N** flags in the floating point status register to reflect the new contents of FPAC.

Computes the effective address, *E*, which addresses a 4-word (double precision) operand.

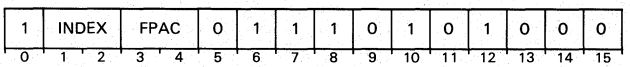
The source operand is checked for a zero mantissa. If the mantissa is zero, the **DVZ** bit is set in the floating point status register and the instruction is terminated. The number in FPAC remains unchanged. If the mantissa is nonzero, the previous contents of FPAC are lost. The two mantissas are compared and if the mantissa of the number in FPAC is greater than or equal to the mantissa of the source operand, the mantissa of the number in FPAC is shifted right one hex digit and the exponent of the number in FPAC is increased by one.

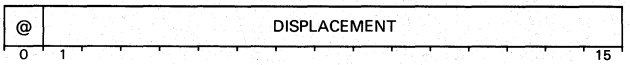
The mantissa in FPAC is then divided by the mantissa of the source operand and the quotient is the mantissa of the intermediate result. The exponent of the source operand is subtracted from the exponent in FPAC and 64 is added to this result. This addition of 64 maintains the *excess 64* notation. The result of the exponent manipulation becomes the exponent of the intermediate result. The sign of the intermediate result is determined from the sign of the two operands by the rules of algebra. The result is placed in FPAC. (Because the operands are assumed to be normalized, and division with such operands produces a normalized result, no normalization of the result takes place.)

If the exponent processing produces either overflow or underflow, the corresponding bit in the floating point status register is set. The number in FPAC is correct except that, for exponent overflow, the exponent is 128 too small, and for exponent underflow, the exponent is 128 too large.

Divide Single (FPAC by Memory)

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





Divides the floating point number in FPAC by the floating point number in the source location and places the normalized result in FPAC. Destroys the previous contents of FPAC, leaves the contents of the source location unchanged, and updates the **Z** and **N** flags in the floating point status register to reflect the new contents of FPAC.

Computes the effective address *E* which addresses a 2-word (single precision) operand.

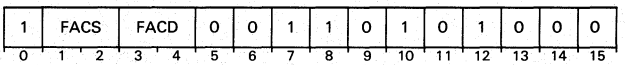
The source operand is checked for a zero mantissa. If the mantissa is zero, the **DVZ** bit is set in the floating point status register and the instruction is terminated. The number in FPAC remains unchanged. If the mantissa is nonzero, the previous contents of FPAC are lost. The two mantissas are compared and if the mantissa of the number in FPAC is greater than or equal to the mantissa of the source operand, the mantissa of the number in FPAC is shifted right one hex digit and the exponent of the number in FPAC is increased by one.

The mantissa in FPAC is then divided by the mantissa of the source operand and the quotient is the mantissa of the intermediate result. The exponent of the source operand is subtracted from the exponent in FPAC and 64 is added to this result. This addition of 64 maintains the *excess 64* notation. The result of the exponent manipulation becomes the exponent of the intermediate result. The sign of the intermediate result is determined from the sign of the two operands by the rules of algebra. The result is placed in FPAC. (Because the operands are assumed to be normalized, and division with such operands produces a normalized result, no normalization of the result takes place.)

If the exponent processing produces either overflow or underflow, the corresponding bit in the floating point status register is set. The number in FPAC is correct except that, for exponent overflow, the exponent is 128 too small, and for exponent underflow, the exponent is 128 too large.

Divide Single (FPAC by FPAC)

FDS *facs,facd*



Divides the floating point number in FACD by the floating point number in FACS and places the normalized result in FACD. Destroys the previous contents of FACD, leaves the contents of FACS unchanged, and updates the **Z** and **N** flags in the floating point status register to reflect the new contents of FACD.

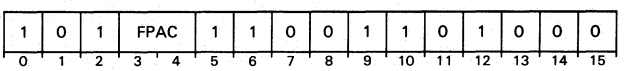
The source operand is checked for a zero mantissa. If the mantissa is zero, the **DVZ** bit is set in the floating point status register and the instruction is terminated. The number in FACD remains unchanged. If the mantissa is nonzero, the previous contents of FACD are lost. The two mantissas are compared, and if the mantissa of the number in FACD is greater than or equal to the mantissa of the source operand, the mantissa of the number in FACD is shifted right one hex digit and the exponent of the number in FACD is increased by one.

The mantissa in FACD is then divided by the mantissa of the source operand and the quotient is the mantissa of the intermediate result. The exponent of the source operand is subtracted from the exponent in FACD and 64 is added to this result. This addition of 64 maintains the *excess 64* notation. The result of the exponent manipulation becomes the exponent of the intermediate result. The sign of the intermediate result is determined from the sign of the two operands by the rules of algebra. The result is placed in FACD. (Because the operands are assumed to be normalized, and division with such operands produces a normalized result, no normalization of the result takes place.)

If the exponent processing produces either overflow or underflow, the corresponding bit in the floating point status register is set. The number in FACD is correct except that, for exponent overflow, the exponent is 128 too small, and for exponent underflow, the exponent is 128 too large.

Load Exponent

FEXP *fpac*



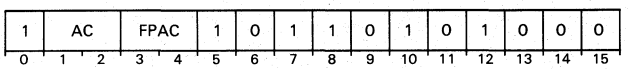
Places bits 1-7 of AC0 in bits 1-7 of the specified FPAC. Ignores bits 0 and 8-15 of AC0. Leaves unchanged bits 0 and 8-63 of FPAC and the entire contents of AC0. Also sets

bits 0-7 (the sign and exponent) to zero if bits 8-63 (the mantissa) of FPAC are zero. Leaves bits 1-7 of FPAC unchanged if FPAC contains true zero.

NOTE: *The exponent contained in bits 1-7 of AC0 is assumed to be in* Excess 64 *representation.*

Fix To AC

FFAS *ac,fpac*



Converts the integer portion of the floating point number contained in the specified FPAC to a signed two's complement integer and places the result in an accumulator.

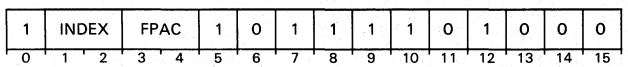
Forms the absolute value of the integer portion of the floating point number in FPAC. Extracts the 15 least significant bits from this value and, if the number in FPAC is negative, forms the two's complement of the integer. Then places the result in the specified accumulator, sets the **Z** and **N** flags in the floating point status register to 0, and leaves the contents of FPAC unchanged.

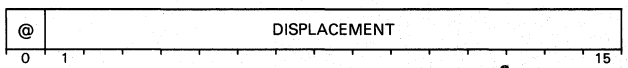
If the number in FPAC is less than -32,768 or greater than +32,767, this instruction sets the **MOF** flag in the floating point status register to 1.

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

Fix To Memory

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





Converts the integer portion of a floating point number to double-precision integer format and stores the result in two memory locations.

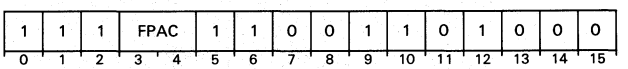
Forms the absolute value of the integer portion of the floating point number in FPAC. Extracts the 31 least significant bits from this value and, if the number in FPAC is negative, forms the two's complement of the integer. Then places the result into the locations addressed by E, sets the **Z** and **N** flags in the floating point status register to 0, and leaves the contents of FPAC unchanged.

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

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

Halve

FHLV *fpac*

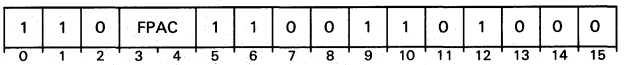


Divides the floating point number in FPAC by 2.

Shifts the mantissa contained in FPAC right one bit position, fills the vacated bit position with a zero and places the bit shifted out in the guard digit. Then normalizes the number and places the result in FPAC. Sets the **UNF** flag in the floating point status register to 1 if the normalization process causes an exponent underflow. The number in FPAC is then correct, except that the exponent is 128 too large. Updates the **Z** and **N** flags in the floating point status register to reflect the new contents of FPAC.

Integerize

FINT *fpac*

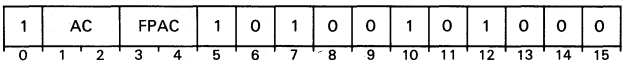


Zeros the fractional portion (if any) of the number contained in the specified FPAC, and then normalizes the number. The instruction updates the **Z** and **N** flags in the floating point status register to reflect the new contents of the specified FPAC.

NOTE: *If the absolute value of the number contained in the specified FPAC is less than 1, the specified FPAC is set to true zero.*

Float From AC

FLAS *ac,fpac*



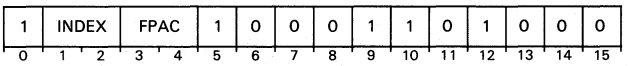
Converts a two's complement number to floating point format.

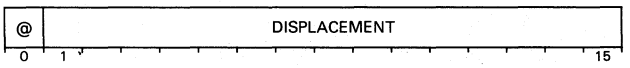
Converts the signed two's complement number contained in the specified accumulator to a single precision floating point number, places the result in the specified FPAC, and sets the low-order 32 bits of the FPAC to 0. Leaves the contents of the specified accumulator unchanged and destroys the previous contents of the FPAC. Updates the **Z** and **N** flags in the floating point status register to reflect the new contents of FPAC.

The range of numbers that can be converted is -32,768 to +32,767.

Load Floating Point Double

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



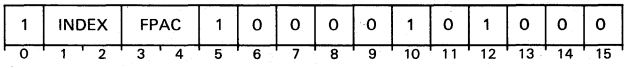


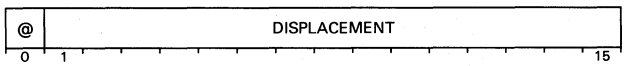
Moves four words out of memory into a specified FPAC.

Computes the effective address, *E*, and places the double precision floating point number at that address in FPAC. Also sets the sign and exponent to zero if the mantissa is zero. Destroys the previous contents of FPAC and updates the **Z** and **N** flags in the FPSR to reflect the new contents of FPAC.

Load Floating Point Single

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



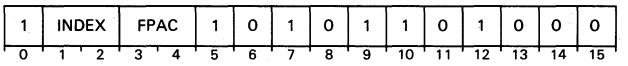


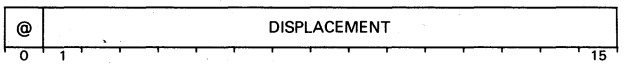
Moves two words out of memory into a specified FPAC.

Computes the effective address *E* and places the single precision floating point number at that address in FPAC. Also sets the sign and exponent to zero if the mantissa is zero. Destroys the previous contents of FPAC and updates the **Z** and **N** flags in the floating point status register to reflect the new contents of FPAC. The low-order 32 bits of FPAC are set to 0.

Float From Memory

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





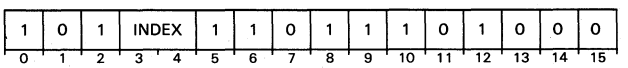
Converts the contents of two memory locations to floating point format and places the result in a specified FPAC.

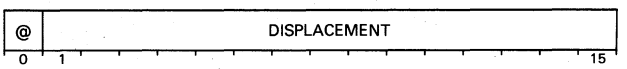
Computes the effective address *E*, converts the 32-bit, signed, two's complement number addressed by *E* to a double precision floating point number, and places the result in the specified FPAC. Destroys the previous contents of FPAC, and updates the **Z** and **N** flags in the floating point status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

The range of numbers that can be converted is -2,147,483,648 to +2,147,483,647.

Load Floating Point Status

FLST *[@]displacement[,index]*



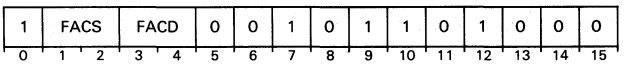


Moves the contents of two specified memory locations to the floating point status register.

Computes the effective address, *E*, places the 32-bit operand addressed by *E* in the floating point status register, and sets the condition codes to the values of the loaded bits.

Multiply Double (FPAC by FPAC)

FMD *facs,facd*



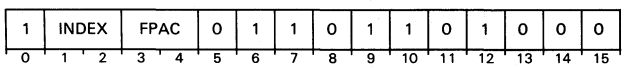
Multiplies the floating point number in FACD by the floating point number in FACS and places the normalized result in FACD. Destroys the previous contents of FACD, leaves the contents of FACS unchanged, and updates the **Z** and **N** flags in the floating point status register to reflect the new contents of FACD.

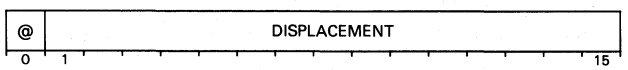
The mantissas of the two numbers are multiplied together to give the mantissa of the intermediate result. One guard digit is provided for the intermediate result, which is used if normalization is required. The exponents of the two numbers are added together and 64 is subtracted. This subtraction of 64 maintains the *excess 64* notation. The result of the exponent manipulation becomes the exponent of the intermediate result. The sign of the intermediate result is determined from the signs of the two operands by the rules of algebra.

If the exponent processing produces either overflow or underflow, the result is held until normalization, as that procedure may correct the condition. If normalization does not correct the condition, the corresponding flag in the floating point status register is set to 1. The number is correct except that, for exponent overflow, the exponent is 128 too small, and for exponent underflow, the exponent is 128 too large.

Multiply Double (FPAC by Memory)

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





Multiplies the floating point number in FPAC by the floating point number in the source location and places the normalized result in FPAC. Destroys the previous contents of FPAC, leaves the contents of the source location unchanged, and updates the **Z** and **N** flags in the floating point status register to reflect the new contents of FPAC.

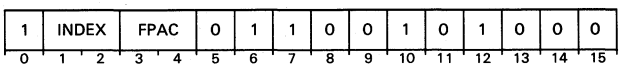
Computes the effective address, *E*, which addresses a 4-word (double precision) operand.

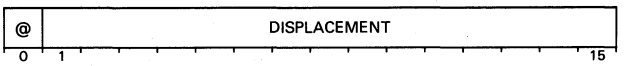
The mantissas of the two numbers are multiplied together to give the mantissa of the intermediate result. One guard digit is provided for the intermediate result, which is used if normalization is required. The exponents of the two numbers are added together and 64 is subtracted. This subtraction of 64 maintains the *excess 64* notation. The result of the exponent manipulation becomes the exponent of the intermediate result. The sign of the intermediate result is determined from the signs of the two operands by the rules of algebra.

If the exponent processing produces either overflow or underflow, the result is held until normalization, as that procedure may correct the condition. If normalization does not correct the condition, the corresponding flag in the floating point status register is set to 1. The number is correct except that, for exponent overflow, the exponent is 128 too small, and for exponent underflow, the exponent is 128 too large.

Multiply Single (FPAC by Memory)

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





Multiplies the floating point number in FPAC by the floating point number in the source location and places the normalized result in FPAC. Destroys the previous contents of FPAC, leaves the contents of the source location unchanged, and updates the **Z** and **N** flags in the floating point status register to reflect the new contents of FPAC.

Computes the effective address *E* which addresses a 2-word (single precision) operand.

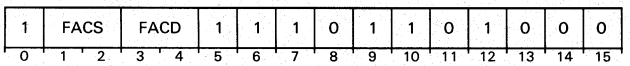
The mantissas of the two numbers are multiplied together to give the mantissa of the intermediate result. One guard digit is provided for the intermediate result, which is used if normalization is required. The exponents of the two numbers are added together and 64 is subtracted. This subtraction of 64 maintains the *excess 64* notation. The result of the exponent manipulation becomes the exponent of the intermediate result. The sign of the intermediate result is determined from the signs of the two operands by the rules of algebra.

If the exponent processing produces either overflow or underflow, the result is held until normalization, as that procedure may correct the condition. If normalization does not correct the condition, the corresponding flag in the

floating point status register is set to 1. The number is correct except that, for exponent overflow, the exponent is 128 too small, and for exponent underflow, the exponent is 128 too large.

Move Floating Point

FMOV

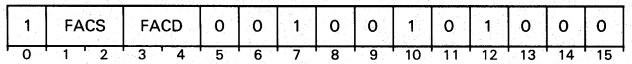


Moves the contents of one FPAC to another FPAC.

Places the contents of FACS in FACD, destroys the previous contents of FACD, and leaves the contents of FACS unchanged. If the mantissa in FACS is zero, the sign and exponent in FACD are also set to zero. The **Z** and **N** flags in the floating point status register are set to reflect the new contents of FACD.

Multiply Single (FPAC by FPAC)

FMS



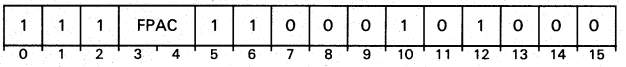
Multiplies the floating point number in FACD by the floating point number in FACS and places the normalized result in FACD. Destroys the previous contents of FACD, leaves the contents of FACS unchanged, and updates the **Z** and **N** flags in the floating point status register to reflect the new contents of FACD.

The mantissas of the two numbers are multiplied together to give the mantissa of the intermediate result. One guard digit is provided for the intermediate result, which is used if normalization is required. The exponents of the two numbers are added together and 64 is subtracted. This subtraction of 64 maintains the *excess 64* notation. The result of the exponent manipulation becomes the exponent of the intermediate result. The sign of the intermediate result is determined from the signs of the two operands by the rules of algebra.

If the exponent processing produces either overflow or underflow, the result is held until normalization, as that procedure may correct the condition. If normalization does not correct the condition, the corresponding flag in the floating point status register is set to 1. The number is correct except that, for exponent overflow, the exponent is 128 too small, and for exponent underflow, the exponent is 128 too large.

Negate

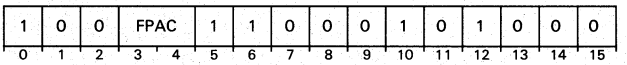
FNEG *fpac*



Inverts the sign bit of FPAC. Bits 1-63 of FPAC remain unchanged. Also sets the sign and exponent to zero if the mantissa in FPAC is zero. Updates the **Z** and **N** flags in the floating point status register to reflect the new contents of FPAC. If FPAC contains true zero, the sign bit remains unchanged.

Normalize

FNOM *fpac*

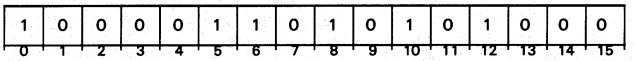


Normalizes the floating point numbers in FPAC. Sets a true zero in FPAC if all the bits of the mantissa are zero. Sets the **UNF** flag in the FPSR if an exponent underflow occurs. The number in FPAC is then correct, except that the exponent is 128 too large.

The **Z** and **N** flags in the floating point status register are set to reflect the new contents of FPAC.

No Skip

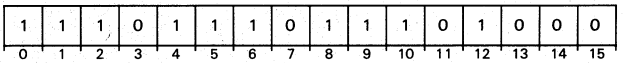
FNS



The next sequential word is executed.

Pop Floating Point State

FPOP



Pops an 18-word floating point return block off the user stack and alters the state of the floating point unit. The words popped and their destinations are diagrammed in Figure 4.3.

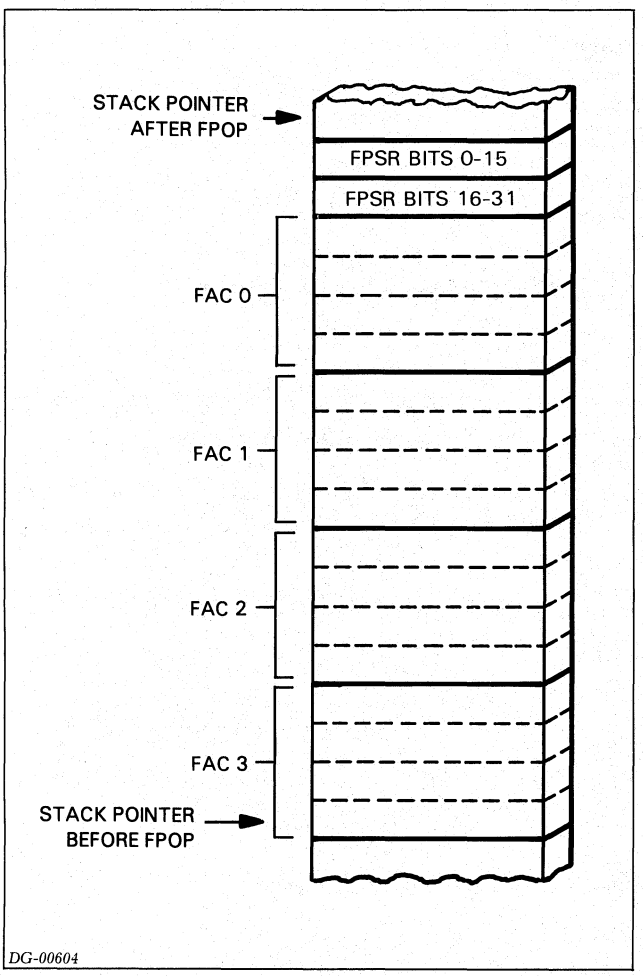


Figure 4.3 Effects of FPOP instruction

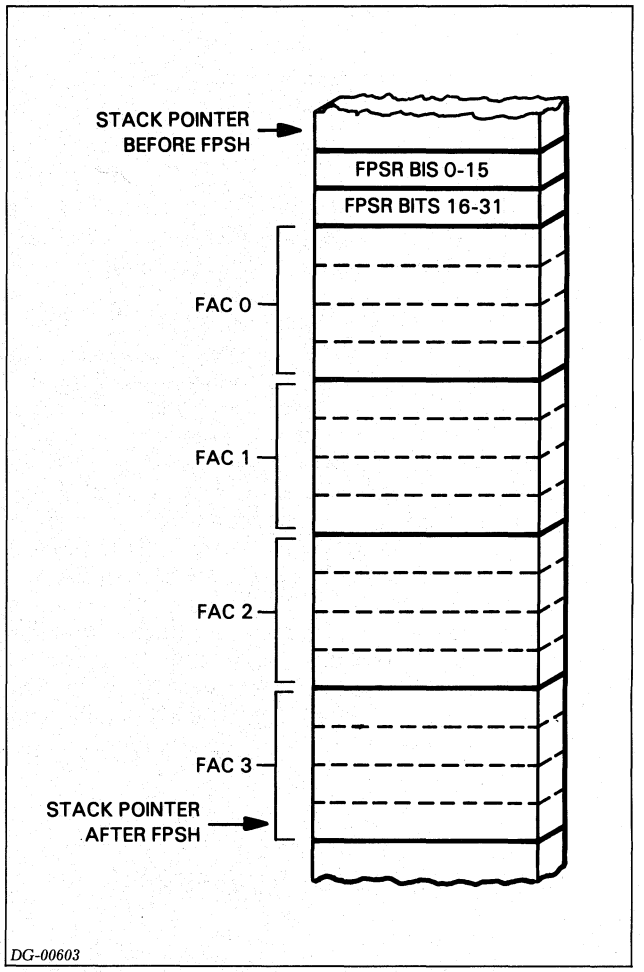


Figure 4.4 Effects of FPSH instruction

NOTE: *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.*

Push Floating Point State

FPSH

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

Pushes an 18-word floating point return block onto the user stack, leaving the contents of the floating point accumulators and the floating point status register unchanged. The format of the 18 words pushed is shown in Figure 4.4.

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 high-order bits of a floating point word to an accumulator.

Places the 16 high-order bits of FPAC into AC0, losing the previous contents of AC0, and leaving unchanged the contents of FPAC and the floating point status register.

Skip Always

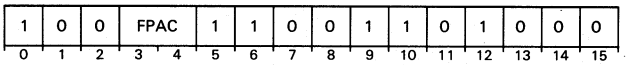
FSA

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

The next sequential word is skipped.

Scale

FSCAL *fpac*

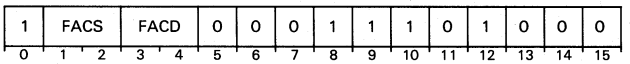


Shifts the mantissa of the floating point number in FPAC either right or left, depending upon the contents of bits 1-7 of AC0. Leaves the contents of AC0 unchanged.

Treats bits 1-7 of AC0 as an exponent in *Excess 64* representation. Computes the difference between this exponent and the exponent in FPAC by subtracting the exponent in FPAC from the number contained in AC0 bits 1-7. If the difference is zero, the instruction stops. If the difference is positive, the instruction shifts the mantissa contained in FPAC right that number of hex digits. If the difference is negative, the instruction shifts the mantissa contained in FPAC left that number of hex digits and sets the **MOF** flag in the floating point status register. After the shift, the contents of bits 1-7 of AC0 replace the exponent contained in FPAC. Bits shifted out of either end of the mantissa are lost. If the entire mantissa is shifted out of FPAC, the instruction sets FPAC to true zero. The instruction sets the **Z** and **N** flags in the floating point status register to reflect the new contents of FPAC.

Subtract Double (FPAC from FPAC)

FSD

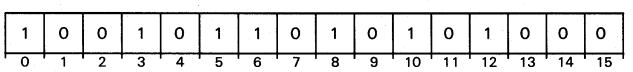


Subtracts the floating point number in FACS from the floating point number in FACD and places the normalized result in the FACD. Destroys the previous contents of FACD, leaves the contents of FACS unchanged, and updates the **Z** and **N** flags in the floating point status register to reflect the new contents of FACD.

The subtraction is performed by inverting the sign bit of the source operand and adding. After the sign inversion, the operation is equivalent to floating point addition. (See **FAD**.)

Skip On Zero

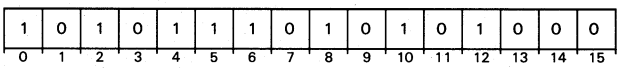
FSEQ



Skips the next sequential word if the **Z** flag of the floating point status register is 1.

Skip On Greater Than Or Equal To Zero

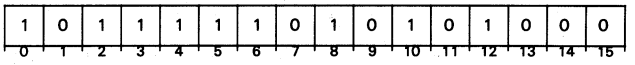
FSGE



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

Skip On Greater Than Zero

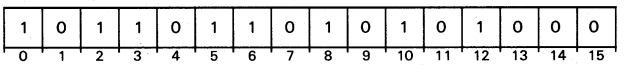
FSGT



Skips the next sequential word if both the **Z** and **N** flags of the floating point status register are 0.

Skip On Less Than Or Equal To Zero

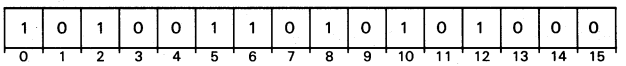
FSLE



Skips the next sequential instruction if either the **Z** flag or the **N** flag of the floating point status register is 1.

Skip On Less Than Zero

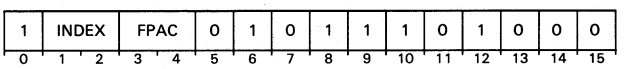
FSLT

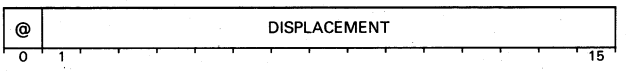


Skips the next sequential word if the **N** flag of the floating point status register is 1.

Subtract Double (Memory from FPAC)

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





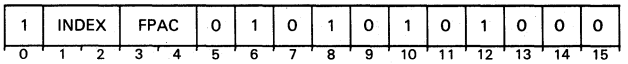
Subtracts the floating point number in the source location from the floating point number in FPAC and places the normalized result in the FPAC. Destroys the previous contents of FPAC, leaves the contents of the source location unchanged, and updates the **Z** and **N** flags in the floating point status register to reflect the new contents of FPAC.

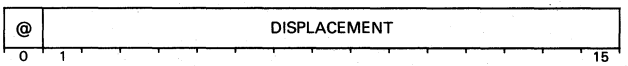
The instruction computes the effective address, *E*, which addresses a 4-word (double precision) operand.

The subtraction is performed by inverting the sign bit of the source operand and adding. After the sign inversion, the operation is equivalent to floating point addition. (See **FAMD**.)

Subtract Single (Memory from FPAC)

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





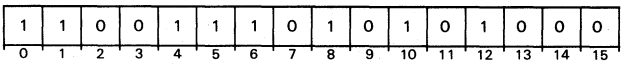
Subtracts the floating point number in the source location from the floating point number in FPAC and places the normalized result in the FPAC. Destroys the previous contents of FPAC, leaves the contents of the source location unchanged, and updates the **Z** and **N** flags in the floating point status register to reflect the new contents of FPAC.

The instruction computes the effective address, *E*, which addresses a 2-word (single precision) operand.

The subtraction is performed by inverting the sign bit of the source operand and adding. After the sign inversion, the operation is equivalent to floating point addition. (See **FAMS**.)

Skip On No Zero Divide

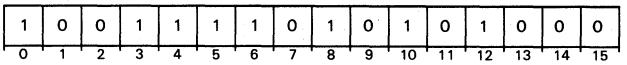
FSND



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

Skip On Non-Zero

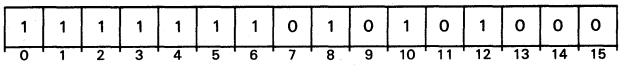
FSNE



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

Skip On No Error

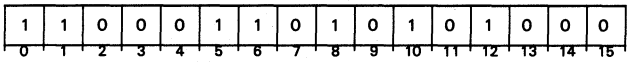
FSNER



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

Skip On No Mantissa Overflow

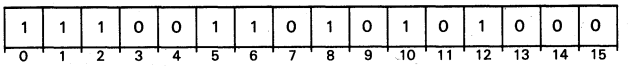
FSNM



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

Skip On No Overflow

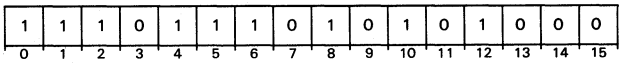
FSNO



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

Skip On No Overflow and No Zero Divide

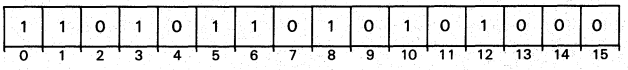
FSNOD



Skips the next sequential word if both the overflow (**OVF**) flag and the divide by zero (**DVZ**) flag of the floating point status register are 0.

Skip On No Underflow

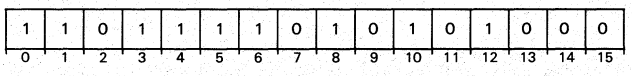
FSNU



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

Skip On No Underflow And No Zero Divide

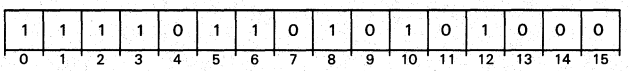
FSNUD



Skips the next sequential word if both the underflow (**UNF**) flag and the divide by zero (**DVZ**) flag of the floating point status register are 0.

Skip On No Underflow And No Overflow

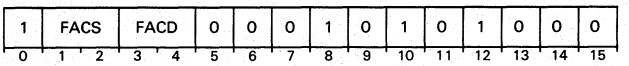
FSNUO



Skips the next sequential word if both the underflow (**UNF**) flag and overflow (**OVF**) flag of the floating point status register are 0.

Subtract Single (FPAC from FPAC)

FSS *facs,facd*

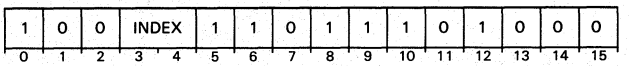


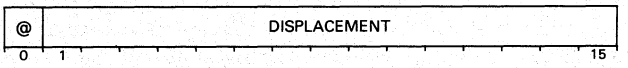
Subtracts the floating point number in FACS from the floating point number in FACD and places the normalized result in the FACD. Destroys the previous contents of FACD, leaves the contents of FACS unchanged, and updates the **Z** and **N** flags in the floating point status register to reflect the new contents of FACD.

The subtraction is performed by inverting the sign bit of the source operand and adding. After the sign inversion, the operation is equivalent to floating point addition.

Store Floating Point Status

FSST *[@]displacement[,index]*



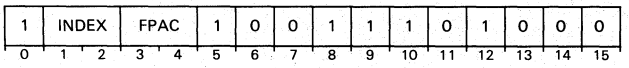


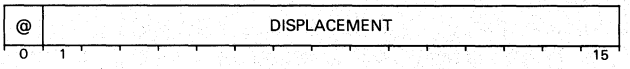
Moves the contents of the FPSR to two specified memory locations.

Computes the effective address, *E*, and places the 32-bit contents of the FPSR in the two consecutive memory locations addressed by *E* and *E* + *1*.

Store Floating Point Double

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



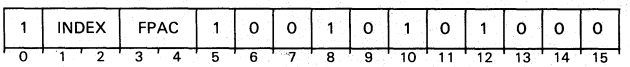


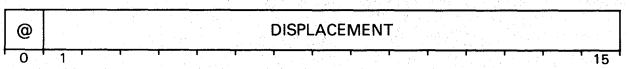
Stores the contents of a specified FPAC into a memory location.

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

Store Floating Point Single

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



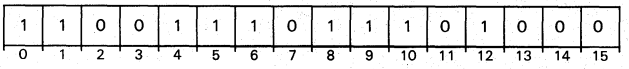


Stores the contents of a specified FPAC into a memory location.

Computes the effective address *E* and places the floating point number contained in FPAC in memory beginning at the location addressed by *E*. Destroys the previous contents of the addressed memory location and leaves unchanged the contents of FPAC and the condition codes in the FPSR. For single precision, only the high-order 32 bits of FPAC are stored.

Trap Disable

FTD

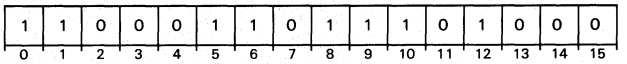


Sets the trap enable bit of the FPSR to 0.

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

Trap Enable

FTE

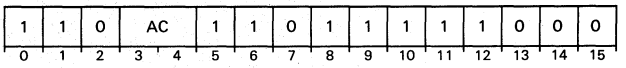


Sets the trap enable bit of the FPSR to 1.

NOTE: *When a floating point fault occurs and the trap enable bit is 1, the trap enable bit is set to 0 before control is transferred to the floating point error handler. The trap enable bit should be set to 1 before normal processing is resumed.*

Halve

HLV *ac*



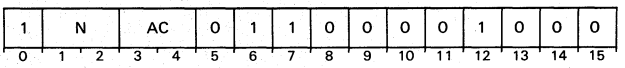
Divides the contents of an accumulator by 2 and rounds the result toward zero.

The signed, 16-bit two's complement number contained in the specified AC is divided by 2 and rounded toward 0. The result is placed in the specified AC.

If the number is positive, division is accomplished by shifting the number right one bit. If the number is negative, division is accomplished by negating the number, shifting it right one bit, and negating it again.

Hex Shift Left

HXL *n,ac*

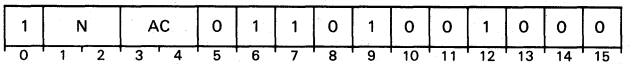


Shifts the contents of AC left a number of hex digits depending upon the immediate field **N**. The number of digits shifted is equal to **N+1.** Bits shifted out are lost, and the vacated bit positions are filled with zeroes. If **N** is equal to 3, then all 16 bits of AC are shifted out and all bits of AC are set to 0.

NOTE: *The assembler takes the coded value of* **N** *and subtracts one from it before placing it in the immediate field. Therefore, you should code the exact number of hex digits that you wish to shift.*

Hex Shift Right

HXR *n,ac*

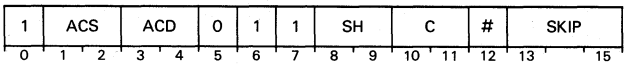


Shifts the contents of AC right a number of hex digits depending upon the immediate field, **N**. The number of digits shifted is equal to **N+1.** Bits shifted out are lost, and the vacated bit positions are filled with zeroes. If **N** is equal to 3, then all 16 bits of AC are shifted out and all bits of AC are set to 0.

> **NOTE:** *The assembler takes the coded value of* **N** *and subtracts one from it before placing it in the immediate field. Therefore, you should code the exact number of hex digits that you wish to shift.*

Increment

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



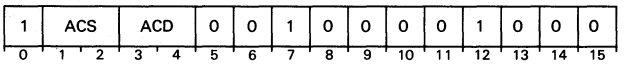
Increments the contents of an accumulator.

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 incrementation produces a carry of 1 out of the high order bit, the instruction complements carry. 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* 177777_8 *the instruction complements carry.*

Inclusive OR

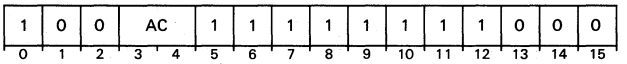
IOR *acs,acd*

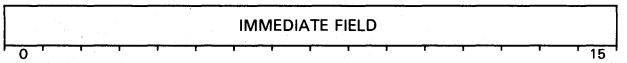


Forms the logical inclusive OR of the contents of ACS and the contents of ACD and places the result in ACD. Sets a bit position in the result to 1 if the corresponding bit position in one or both operands contains a 1; otherwise, the instruction sets the result bit to 0. The contents of ACS remain unchanged.

Inclusive OR Immediate

IORI *i,ac*

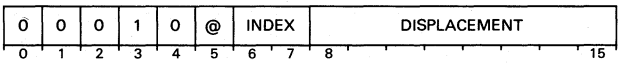




Forms the logical inclusive OR of the contents of the immediate field and the contents of the specified AC and places the result in the specified AC.

Increment And Skip If Zero

ISZ *[@]displacement[,index]*

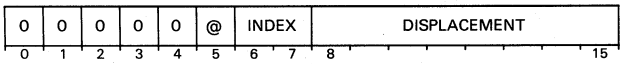


Increments the addressed word, then skips if the incremented value is zero.

Increments the word addressed by *E* and writes the result back into memory at that location. If the updated value of the location is zero, the instruction skips the next sequential word.

Jump

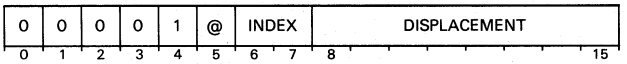
JMP



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.

Jump To Subroutine

JSR *[@]displacement[,index]*



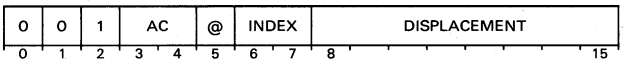
Increments and stores the value of the program counter in AC3, and then places a new address in the program counter.

Computes the effective address, *E*; then places the address of the next sequential instruction in AC3. Places *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.*

Load Accumulator

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

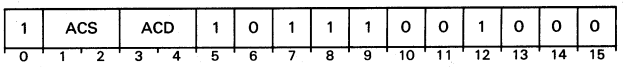


Copies a word from memory to an accumulator.

Places the word addressed by the effective address, *E*, in the specified accumulator. The previous contents of the location addressed by *E* remain unchanged.

Load Byte

LDB *acs,acd*

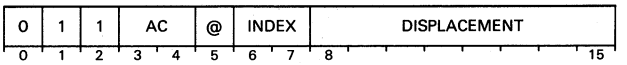


Moves a byte from memory (as addressed by a byte pointer in one accumulator) to the second accumulator.

Places the 8-bit byte addressed by the byte pointer contained in ACS in 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.

Load Effective Address

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



Computes the effective address, *E*, and places it in bits 1-15 of the specified accumulator. Sets bit 0 of the accumulator to 0. The previous contents of the AC are lost.

If you reference an auto-incrementing or auto-decrementing location during the effective address calculation, the instruction increments or decrements as appropriate the contents of the auto-incrementing or -decrementing location.

Figure 4.5 shows some different uses of the **LEF** instruction.

```
LEF     0,TABLE     ;The logical address of
                    ;TABLE is placed in AC0.
LEF     1,-55,3     ;Subtracts 000055 (octal)
                    ;from the unsigned integer
                    ;in AC3 and the result is
                    ;placed in AC1.
LEF     0, . +0     ;Places the address of this
                    ;Load effective address
                    ;instruction in AC0.
```

DG-06563

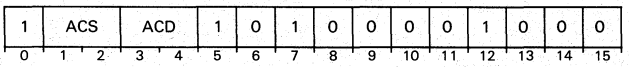
Figure 4.5 Examples of LEF instruction

NOTE: *The* **LEF** *instruction can only be used in a mapped system, while in the user mode. With the* Lef *mode bit set to 1, all I/O and* **LEF** *instructions will be interpreted as* **LEF** *instructions.*

Be sure that I/O protection is enabled or the Lef *mode bit is set to 1 before using the* **LEF** *instruction. If you issue a* **LEF** *instruction in the I/O mode, with protection disabled, the instruction will be interpreted and executed as an I/O instruction, with possibly undesirable results.*

Locate Lead Bit

LOB *acs,acd*

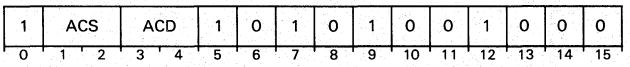


Adds a number equal to the number of high-order zeroes in the contents of ACS to the signed, 16-bit, two's complement number contained in ACD. The contents of ACS and the state of carry remain unchanged.

NOTE: *If ACS and ACD are specified as the same accumulator, the instruction functions as described above, except that since ACS and ACD are the same accumulator, the contents of ACS will be changed.*

Locate and Reset Lead Bit

LRB *acs,acd*



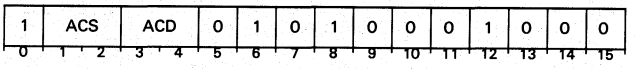
Performs a *Locate Lead Bit* instruction, and sets the lead bit to 0.

Adds a number equal to the number of high-order zeroes in the contents of ACS to the signed, 16-bit, two's complement number contained in ACD. Sets the leading 1 in ACS to 0. The state of carry remains unchanged.

NOTE: *If ACS and ACD are specified to be the same accumulator, then the instruction sets the leading 1 in that accumulator to 0, and no count is taken.*

Logical Shift

LSH *acs,acd*



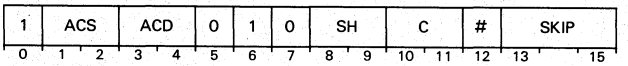
Shifts the contents of ACD either left or right depending on the number contained in bits 8-15 of ACS. The signed, 8-bit two's complement number contained in bits 8-15 of ACS determines the direction of the shift and the number of bits to be shifted. If the number in bits 8-15 of ACS is positive, shifting is to the left; if the number in bits 8-15 of ACS is negative, shifting 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.

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 zeroes. The carry bit and the contents of ACS remain unchanged.

NOTE: *If the magnitude of the number in bits 8-15 of ACS is greater than 15, all bits ACD are set to 0. The carry bit and the contents of ACS remain unchanged.*

Move

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

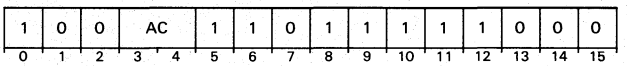


Moves the contents of an accumulator through the Arithmetic Logic Unit (ALU).

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.

Modify Stack Pointer

MSP *ac*



Changes the value of the stack pointer and tests for potential overflow.

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

value of the stack limit. If the result in location 40 is less than the stack limit, then the instruction ends.

If the result is greater than the stack limit, the instruction changes the value of location 40 back to its original contents before the add. The instruction pushes a return block of the format shown below:

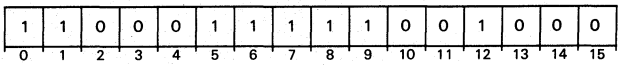
Word Pushed	Contents
1	AC0
2	AC1
3	AC2
4	AC3
5	Bit 0 equals carry. Bits 1-15 equal PC; contain address of *Modify Stack Pointer* instruction.

The program counter in the return block contains the address of the *Modify Stack Pointer* instruction.

After pushing the return block, the program counter contains the address of the stack fault routine. The stack pointer is updated with the value used to push the return block, and control transfers to the stack fault routine.

Unsigned Multiply

MUL

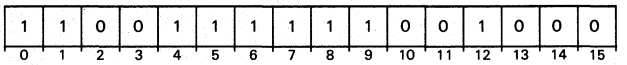


Multiplies the unsigned contents of two accumulators and adds the result to the unsigned contents of a third accumulator. The result is an unsigned 32-bit integer in two accumulators.

The unsigned, 16-bit number in AC1 is multiplied 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 and occupies AC0 and AC1. Bit 0 of AC0 is the high-order bit of the result and bit 15 of AC1 is the low-order bit. The contents of AC2 remain unchanged. Because the result is a double-length number, overflow cannot occur.

Signed Multiply

MULS



Multiplies the signed contents of two accumulators and adds the result to the signed contents of a third accumulator. The result is a signed 32-bit integer in two accumulators.

The signed, 16-bit two's complement number in AC1 is multiplied by the signed, 16-bit two's complement number in AC2 to yield a signed, 32-bit two's complement intermediate result. The signed, 16-bit two's complement number in AC0 is added to the intermediate result to produce the final result. The final result is a signed, 32-bit two's complement number which occupies AC0 and AC1. Bit 0 of AC0 is the sign bit of the result and bit 15 of AC1 is the low-order bit. The contents of AC2 remain unchanged. Because the result is a double-length number, overflow cannot occur.

Negate

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

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

Forms the two's complement of the contents of an accumulator.

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 out of the high-order 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.*

Pop Multiple Accumulators

POP *acs,acd*

1	ACS	ACD	1	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

Pops 1 to 4 words off the stack and places them in the indicated accumulators.

The set of accumulators from ACS through ACD is filled with words popped from the stack. The accumulators are filled in descending order, starting with the AC specified by ACS and continuing down through the AC specified by ACD, wrapping around if necessary, with AC3 following AC0. If ACS is equal to ACD, only one word is popped and it is placed in ACS.

The stack pointer is decremented by the number of accumulators popped and the frame pointer is unchanged. A check for underflow is made only after the entire pop operation is done.

Pop Block

POPB

1	0	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

Returns control from a *System Call* routine or an I/O interrupt handler that does not use the stack change facility of the *Vector* instruction.

Five words are popped off the stack and placed in predetermined locations. The words popped and their destinations are shown in Figure 4.6.

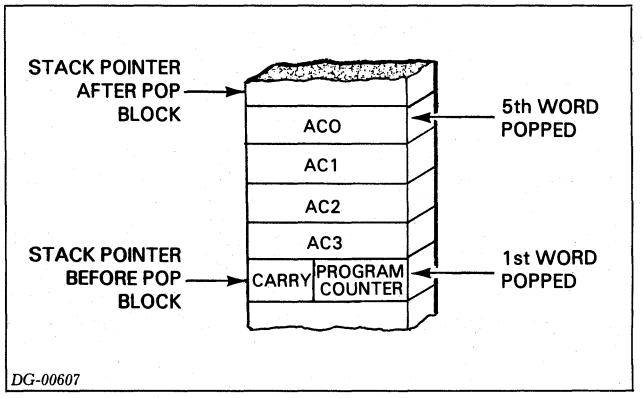


Figure 4.6 Effects of POPB instruction

Sequential operation is continued with the word addressed by the updated value of the program counter.

NOTE: *If the I/O handler uses the stack change facility of the* Vector on Interrupting Device Code *instruction, do not use the* Pop Block *instruction. Use the* Restore *instruction instead.*

Pop PC And Jump

POPJ

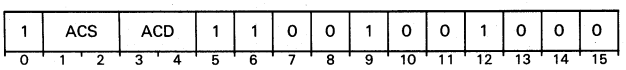
1	0	0	1	1	1	1	1	1	1	0	0	1	0	0	0
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

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

The stack pointer is decremented by one and the frame pointer is unchanged. A check for underflow occurs after the pop operation.

Push Multiple Accumulators

PSH *acs,acd*



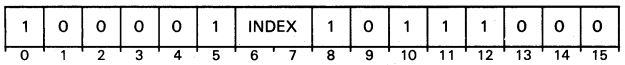
Pushes the contents of 1 to 4 accumulators onto the stack.

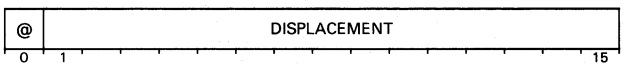
The set of accumulators from ACS through ACD is pushed onto the stack. The accumulators are pushed in ascending order, starting with the AC specified by ACS and continuing up through the AC specified by ACD, wrapping around if necessary, with AC0 following AC3. The contents of the accumulators remain unchanged. If ACS equals ACD, only ACS is pushed.

The stack pointer is incremented by the number of accumulators pushed and the frame pointer is unchanged. A check for overflow is made only after the entire push operation finishes.

Push Jump

PSHJ *[@]displacement[,index]*

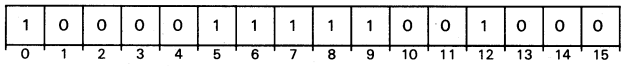




Pushes the address of the next sequential instruction onto the stack, computes the effective address, *E*, and places it in the program counter. Sequential operation continues with the word addressed by the updated value of the program counter.

Push Return Address

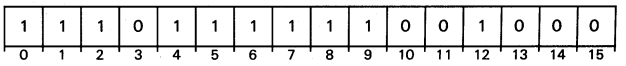
PSHR



Pushes the address of this instruction *plus 2* onto the stack.

Restore

RSTR



Returns control from certain types of I/O interrupts.

Pops nine words off the stack and places them in predetermined locations. The words popped and their destinations are shown in Figure 4.7.

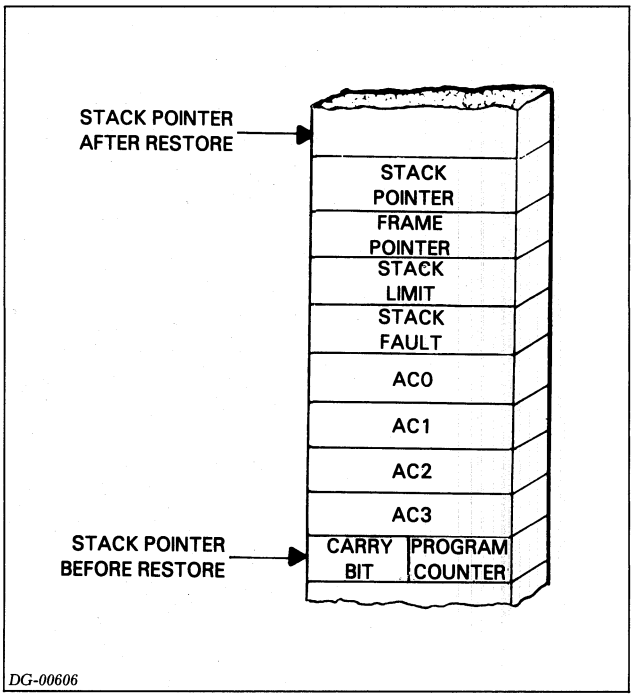


Figure 4.7 Effects of RSTR instruction

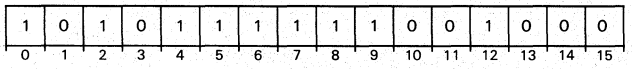
Sequential operation continues with the word addressed by the updated value of the program counter.

NOTE: *Use the* Restore *instruction to return control to the program only if the I/O interrupt handler uses the stack change facility of the* Vector on Interrupting Device Code *instruction.*

The Restore *instruction does not check for stack underflow.*

Return

RTN



Returns control from subroutines that issue a *Save* instruction at their entry points.

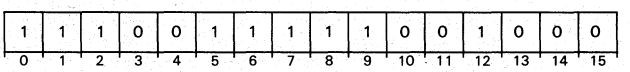
The *Save* instruction loads the current value of the stack pointer into the frame pointer. The *Return* instructions uses this value of the frame pointer to pop a standard return block off of the stack. The format of the return block is shown below.

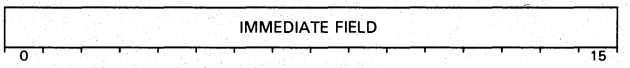
Word Popped	Destination
1	Bit 0 is loaded into carry. Bits 1-15 are loaded into the PC.
2	AC3
3	AC2
4	AC1
5	AC0

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.

Save

SAVE *i*





Saves the information required by the *Return* instruction.

Saves the current value of the stack pointer in a temporary location. Adds five plus the unsigned, 16-bit integer contained in the immediate field to the current value of the stack pointer and loads the result into location 40. Compares this new value of the stack pointer to the stack limit to check for overflow. If no overflow condition exists, then the instruction places the current value of the frame pointer in AC3. Fetches the contents of the temporary location and loads them into the frame pointer. The instruction uses the value in the frame pointer to push a five-word return block. The formats and contents of the five-word return block is as follows:

Word Pushed	Contents
1	AC0
2	AC1
3	AC2
4	Frame pointer before the save.
5	Bit 0 = carry bit. Bits 1-15 = bits 1-15 of AC3.

After pushing the return block, the instruction places the value of the frame pointer (which now contains the old value of the stack pointer plus five) in AC3.

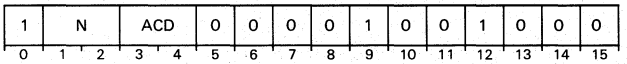
If an overflow condition exists, the *Save* instruction transfers control to the stack fault routine. The program counter in the fault return block contains the address of the *Save* instruction.

The *Save* instruction allocates a portion of the stack for use by the procedure which executed the *Save*. The value of the *frame size*, contained in the immediate field, determines the number of words in this stack area. This portion of the stack will not normally be accessed by push and pop operations, but will be used by the procedure for temporary storage of variables, counters, etc. The frame pointer acts as the reference point for this storage area.

Use the *Save* instruction with the *Jump to Subroutine* instruction. The *Jump to Subroutine* instruction places the return value of the program counter in AC3. *Save* then pushes the return value (contents of AC3) into bits 1-15 of the fifth word pushed.

Subtract Immediate

SBI *n,ac*



Subtracts an unsigned integer in the range 1-4 from the contents of an accumulator.

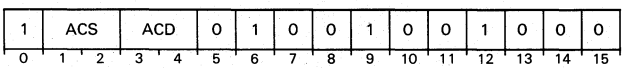
The contents of the immediate field *N*, plus 1 are subtracted from the unsigned 16-bit number contained in the specified AC and the result is placed in ACD. Carry remains unchanged.

> **NOTE:** *The assembler takes the coded value of* n *and subtracts one from it before placing it in the immediate field. Therefore code the exact value you wish to subtract.*

— Assume that AC2 contains 000003_8. After the instruction **SBI 4,2** is executed, AC2 contains 177777_8 and carry remains unchanged.

Skip If ACS Greater Than Or Equal to ACD

SGE *acs,acd*



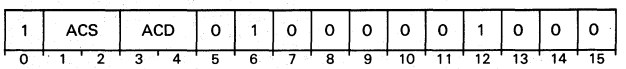
Compares two signed integers in two accumulators and skips if the first is greater than or equal to the second.

The signed two's complement numbers in ACS and ACD are algebraically compared. 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 ACD remain unchanged.

> **NOTE:** *The* Skip If ACS Greater Than ACD *and* Skip If ACS Greater Than Or Equal To ACD *instructions treat the contents of the specified accumulators as signed, two's complement integers. To compare unsigned integers, use the* Subtract *and* Add Complement *instructions.*

Skip If ACS Greater Than ACD

SGT *acs,acd*

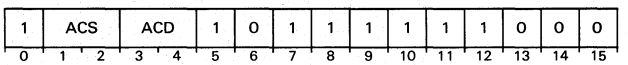


Compares two signed integers in two accumulators and skips if the first is greater than the second.

The signed, two's complement numbers in ACS and ACD are algebraically compared. 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.

Skip On Non-Zero Bit

SNB *acs,acd*



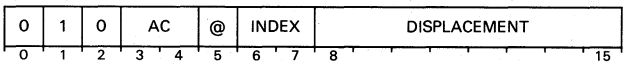
The two accumulators form a bit pointer. If the addressed bit is 1, the next sequential word is skipped.

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

If the addressed bit in memory is 1, the next sequential word is skipped. The contents of ACS and ACD remain unchanged.

Store Accumulator

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

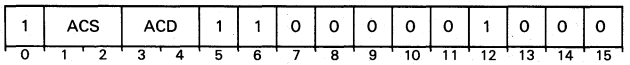


Stores the contents of an accumulator into a memory location.

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

Store Byte

STB *acs,acd*

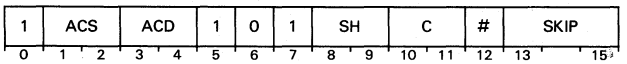


Moves the right byte of ACD to a byte in memory. ACS contains the byte pointer.

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.

Subtract

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



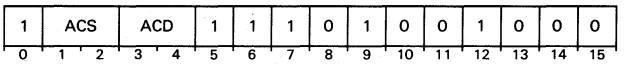
Performs unsigned integer subtraction and complements carry if appropriate.

Initializes carry to its specified value. The instruction subtracts the unsigned, 16-bit number in 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. The instruction places the result of the addition in the shifter. If the operation produces a carry of 1 out of the high-order bit, the instruction complements carry. The instruction 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 instruction 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.*

System Call

SYC *acs,acd*



Pushes a return block and transfers control to the *system call handler.*

If a user map is enabled, the instruction disables it and pushes a return block onto the stack. The program counter in the return block points to the instruction immediately following the *System call* instruction. After pushing the return block, the instruction executes a *jump indirect* to location 2, which contains the address of the *system call handler.*

If this instruction disables a user map, then I/O interrupts cannot occur between the time the *System call* instruction is executed and the time the first instruction of the system call handler is executed.

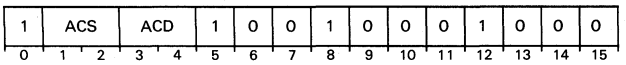
NOTE: *If both accumulators are specified as AC0, the instruction does not push a return block onto the stack. The contents of AC0 remain unchanged.*

The assembler recognizes the mnemonic **SCL** *as equivalent to* **SYC 1,1**.

The assembler recognizes the mnemonic **SVC** *as equivalent to* **SYC 0,0**.

Skip On Zero Bit

SZB *acs,acd*



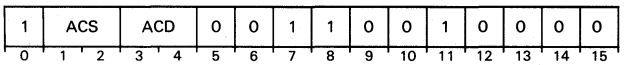
The two accumulators form a bit pointer. If the addressed bit is zero, the next sequential word is skipped.

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

If the addressed bit in memory is 0, the next sequential word is skipped. The contents of ACS and ACD remain unchanged. memory references.")

Skip On Zero Bit And Set To One

SZBO *acs,acd*



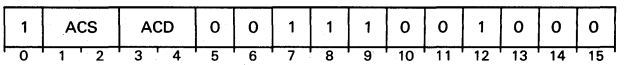
The two accumulators form a bit pointer. The instruction sets the addressed bit to 1. If the addressed bit was 0 before being set to 1, the instruction skips the next sequential word. The contents of ACS and ACD remain unchanged.

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

> **NOTE:** *This instruction facilitates the use of bit maps for such purposes as allocation of facilities (memory blocks, I/O devices, etc.) to several processes, or tasks, that may interrupt one another, or in a multiprocessor environment. The bit is tested and set to 1 in one memory cycle.*

Exchange Accumulators

XCH *acs,acd*

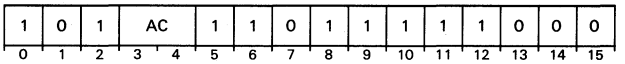


Exchanges the contents of two accumulators.

Places the original contents of ACS in ACD and the original contents of ACD in ACS.

Execute

XCT *ac*



Executes the instruction contained in AC as if it were in main memory in the location occupied by the *Execute* instruction. If the instruction in AC is an *Execute* instruction which executes the instruction in AC, the processor is placed in a one-instruction loop. The Reset switch will stop the processor.

Because of the possibility of AC containing an *Execute* instruction, this instruction is interruptible. An I/O interrupt can occur immediately prior to each time the instruction in AC is executed. If an I/O interrupt does occur, the program counter in the return block pushed on the system stack points to the *Execute* instruction in main memory. This capability to execute an *Execute* instruction gives you a *wait for I/O interrupt* instruction.

> **NOTE:** *If the specified accumulator contains the first word of a two-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. See Figure 4.8 for an example.*

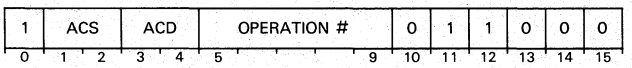
```
        LDA     0,TOT
        XCT     0           ;UNDEFINED
        JMP     ON
TOT:    ADD     1,0
```

DG-06564

Figure 4.8 Example of XCT instruction

Extended Operation

XOP *acs,acd,operation #*



Pushes a return block onto the stack. Places ACS's stack address in AC2; places ACD's stack address in AC3. Memory location 44_8 must contain the XOP origin address, the starting address of a 32_{10} word table of addresses. These addresses are the starting location 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. The instruction fetches that word and treats it as the intermediate address in the effective address calculation. After the indirection chain, if any, has been followed, the instruction places the effective address in the program counter. The contents of AC0, AC1, and the XOP origin address remain unchanged.

The format of the return block pushed by the XOP instruction is shown in Figure 4.9.

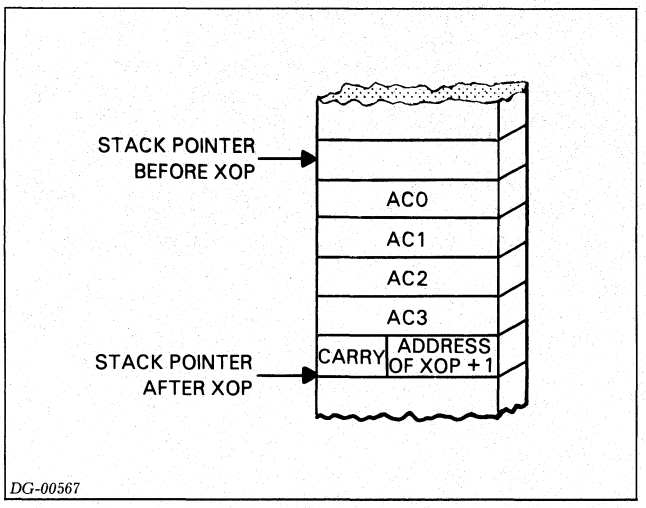
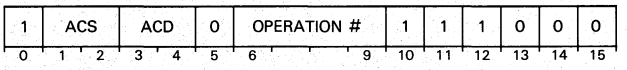


Figure 4.9 Effects of XOP instruction

This return block is configured so that the XOP procedure can return control to the calling program via the *Pop Block* instruction.

Alternate Extended Operation

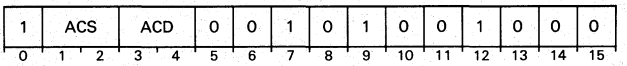
XOP1 *acs,acd,operation #*



This instruction operates exactly like the *Extended Operation* instruction except that it adds 32_{10} to the entry number before it adds the entry number to the XOP origin address. In addition, it can specify only 16 entry locations.

Exclusive OR

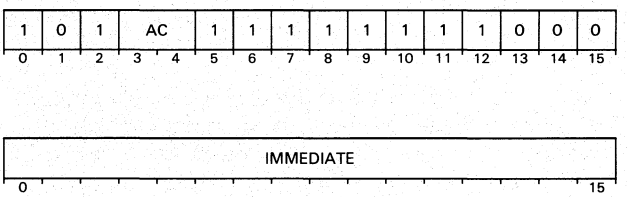
XOR *acs,acd*



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 are unlike; otherwise, the instruction sets result bit to 0. The contents of ACS remain unchanged.

Exclusive OR Immediate

XORI *i,ac*



Forms the logical exclusive OR of the contents of the immediate field and the contents of the specified AC and places the result in the specified AC.

Chapter 5
I/O Instructions

This chapter lists the ECLIPSE S/140 I/O instructions and the special CPU instructions. It also contains instructions intended for specific devices, such as the BMC, ERCC, the MMPU, and the real-time clock. We have arranged the instructions alphabetically within specific I/O device categories. As usual, instructions are alphabetized by the mnenomic recognized by the assembler.

For each instruction we include:

- The mnemonic recognized by the assembler.
- The bit format required.
- The format for any arguments involved.
- The functional description of each instruction.

In general, I/O instructions can be executed only when both *Lef* mode and I/O protection are disabled. (See the *Memory Management and Protection Unit* section in Chapter 3 for a discussion of *Lef* mode and I/O protection.)

General I/O Instructions

You can use the following general I/O instructions with any I/O device, using the appropriate device code.

Device Flag Commands

f=**S** Issues a Start pulse to the specified device.

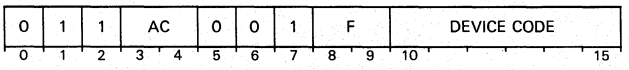
f=**C** Issues a Clear pulse to the specified device.

f=**P** Issues an I/O pulse to the specified device.

IORST No effect.

Data In A

DIA*[f]* *ac,device*



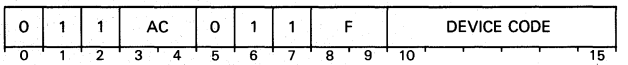
Transfers data from the A buffer of an I/O device to an accumulator.

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

The number of data bits moved depends upon the size of the buffer and the mode of operation of the device. Bits in the AC that do not receive data are set to 0.

Data in B

DIB*[f]* *ac,device*



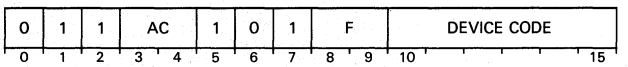
Transfers data from the B buffer of an I/O device to an accumulator.

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

The number of data bits moved depends upon the size of the buffer and the mode of operation of the device. Bits in the AC that do not receive data are set to 0.

Data In C

DIC*[f]* *ac,device*



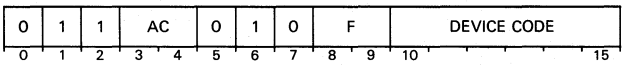
Transfers data from the C buffer of an I/O device to an accumulator.

Places the contents of the C input buffer in the specified device in the specified AC. After the data transfer, sets the Busy and Done flags according to the specified *F*.

The number of data bits moved depends upon the size of the buffer and the mode of operation of the device. Bits in the AC that do not receive data are set to 0.

Data Out A

DOA*[f] ac,device*



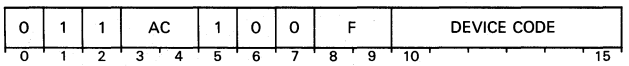
Transfers data from an accumulator to the A buffer of an I/O device.

Places the contents of the specified AC 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 AC remain unchanged.

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

Data Out B

DOB*[f] ac,device*



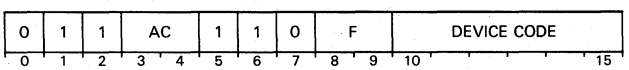
Transfers data from an accumulator to the B buffer of an I/O device.

Places the contents of the specified AC 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 AC remain unchanged.

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

Data Out C

DOC*[f] ac,device*



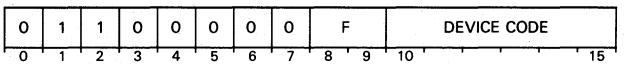
Transfers data from an accumulator to the C buffer of an I/O device.

Places the contents of the specified AC 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 AC remain unchanged.

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

No I/O Transfer

NIO *[f] ac,device*

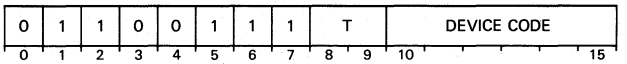


Used when a Busy or Done flag must be changed with no other operation taking place.

Sets the Busy and Done flags in the specified device according to the function specified by *F*.

I/O Skip

SKP*[t] device*



If the test condition specified by *T* is true, the instruction skips the next sequential word.

Central Processor

Device Code - 77_8 (Primary)

Priority Mask Bit - None

Device Flag Commands

Device flag commands to the CPU determine whether the current program can be interrupted by a program interrupt request. When the interrupt on flag is set to 1, the program can be interrupted (once the instruction following the enable has begun). When the interrupt on flag is set to 0, the program cannot be interrupted. The CPU interrupt on flag is controlled by the device flag commands as follows:

f=**S** Sets the interrupt on flag to 1.

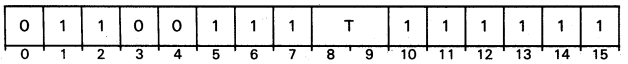
f=**C** Sets the interrupt on flag to 0.

f=**P** If not an **INTA** instruction, no effect. If the instruction is an **INTA** instruction, interprets the **INTA** instruction as the first word of a *Vector* instruction.

IORST Sets the interrupt on flag to 0.

CPU Skip

SKP*[t]* CPU



If the test condition specified by **T** is true, the next sequential word is skipped.

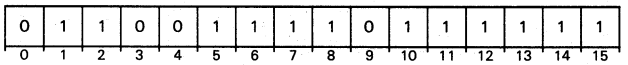
The following table lists the possible test conditions.

Symbol	Value	Test
[t]=**BN**	00	Tests Interrupt On flag for 1
[t]=**BZ**	01	Tests Interrupt On flag for 0
[t]=**DN**	10	Tests Power Fail flag for 1
[t]=**DZ**	11	Tests Power Fail flag for 0

See *Programmer's Reference-Peripherals* (DGC No. 014-000632) for a complete set of examples on using the interrupt system.

CPU Skip If Power Fail Flag Is One

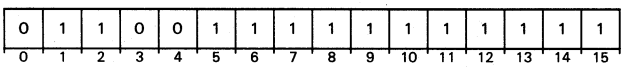
SKPDN CPU



If the Power Fail flag is 1 (i.e., power is failing), the instruction skips the next sequential word.

CPU Skip If Power Fail Flag Is Zero

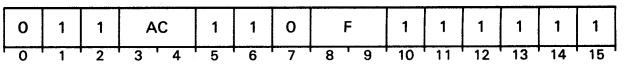
SKPDZ CPU



If the Power Fail flag is 0 (i.e., power is not failing), the instruction skips the next sequential word.

Halt

DOC*[f]* *ac*,CPU



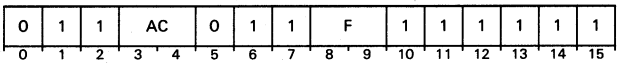
Stops the processor.

Sets the Interrupt On flag according to the function specified by *F*, then stops the processor. The data lights display the contents of the specified accumulator.

> **NOTE:** *The assembler recognizes the special mnemonic* **HALT** *as equivalent to the instruction* **DOC 0,CPU.**

Interrupt Acknowledge

INTA
DIB*[f]* *ac*,CPU



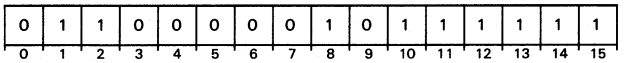
Returns device code of an interrupting device.

Places the six-bit device code of that device requesting an interrupt which is physically closest to the CPU on the I/O bus in bits 10-15 of the specified accumulator; sets bits 0-9 to 0. After the transfer, sets the Interrupt On flag according to the function specified by *F*.

Power fail has the lowest priority for this instruction.

Interrupt Disable

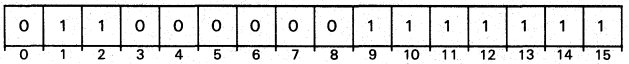
INTDS
NIOC CPU



Sets Interrupt On flag to 0.

Interrupt Enable

INTEN *1*
NIOS CPU

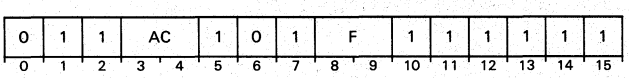


Sets Interrupt On flag to 1.

If the instruction changes the state of the Interrupt On flag, the CPU 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.

Reset

IORST
DIC*[f]* *ac*,**CPU**



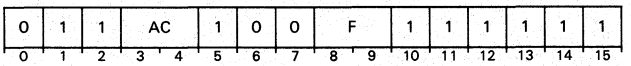
Sets all Busy and Done flags and the priority mask to 0.

Sets the Busy and Done flags in all I/O devices to 0. Sets the 16-bit priority mask to 0. Sets the Interrupt On flag according to the function specified by *F*. Disables the MMPU and clears Map Status Register. All other functions are device dependent.

> **NOTE:** *The assembler recognizes the mnemonic* **IORST** *as equivalent to the instruction* **DICC 0,CPU**.
>
> *If the mnemonic* **DIC** *is used to perform this function, you must code an accumulator to avoid assembly errors. During execution, the accumulator field is ignored and the contents of the accumulator remain unchanged.*

Mask Out

MSKO
DOB*[f]* *ac*,**CPU**



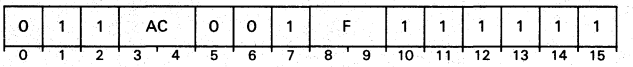
Sets the priority mask.

Places the contents of the specified accumulator in the priority mask. After the transfer, sets the Interrupt On flag according to the function specified by *F*. The contents of the specified AC remain unchanged.

> **NOTE:** *A 1 in any bit disables interrupt requests for those devices which use that bit as a mask.*
>
> **NOTE:** *Do not use this instruction when interrupts are enabled.*

Read Switches

READS *ac*
DIA*[f]* *ac*,**CPU**

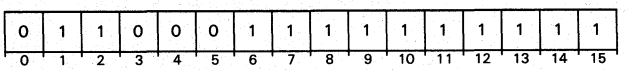


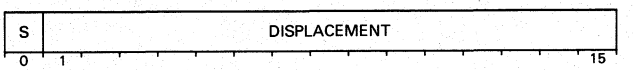
Places the contents of the virtual console switch register into the specified accumulator. After the transfer, sets the Interrupt On flag according to the function specified by F.

Vectored I/O Instruction

Vector On Interrupting Device Code

VCT *[@]displacement[,index]*





Returns the device code of the interrupting device and uses that code as an index into a table. The value found in the table is used in one of two ways: it can be a pointer to the appropriate interrupt handler (Mode A), or as a pointer to another table (Modes B through E). This second table points to the interrupt handler and contains a new priority mask. These operations are shown in Figure 5.1. Depending on the mode used, the instruction can also save the state of the machine by pushing certain information onto the stack, create a new vector stack, set up a priority structure, and enable interrupts. Obviously, the complexity of an operation affects the instruction execution time.

The flow chart in Figure 5.2 is a complete diagram of the operation of the *Vector* instruction. Note that all modes use the *vector table* to find the next address used. Mode A uses the vector table entry as the address of the interrupt

handler and passes control to it immediately. Modes B through E all use the vector table address as a pointer into a *device control table (DCT)*, where the address of the interrupt handler is found, along with a new priority mask.

Three control bits determine the mode of the *Vector* instruction. The names and locations of these bits are:

Stack Change Bit (S) — Bit 0 of the second word of the *Vector* instruction.

Direct Bit (D) — Bit 0 of the selected vector table entry.

Push Bit (P) — Bit 0 of the first word of the selected device control table.

The values of these bits collectively determine the mode of the *Vector* instruction. The table below illustrates mode determination.

Direct	Stack	Push	Mode
0	—	—	A
1	0	0	B
1	0	1	C
1	1	0	D
1	1	1	E

All modes perform the initial steps of the *Vector* instruction. These steps 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 (bits 1-15 of the second instruction word). The result is the address of an entry within the vector table. The instruction 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 through E) is selected.

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 the interrupt handler with all interrupts disabled.

Modes B Through E

Modes B through E perform different functions initially, but use a common second part. The following section discusses the common second part after discussing each mode separately.

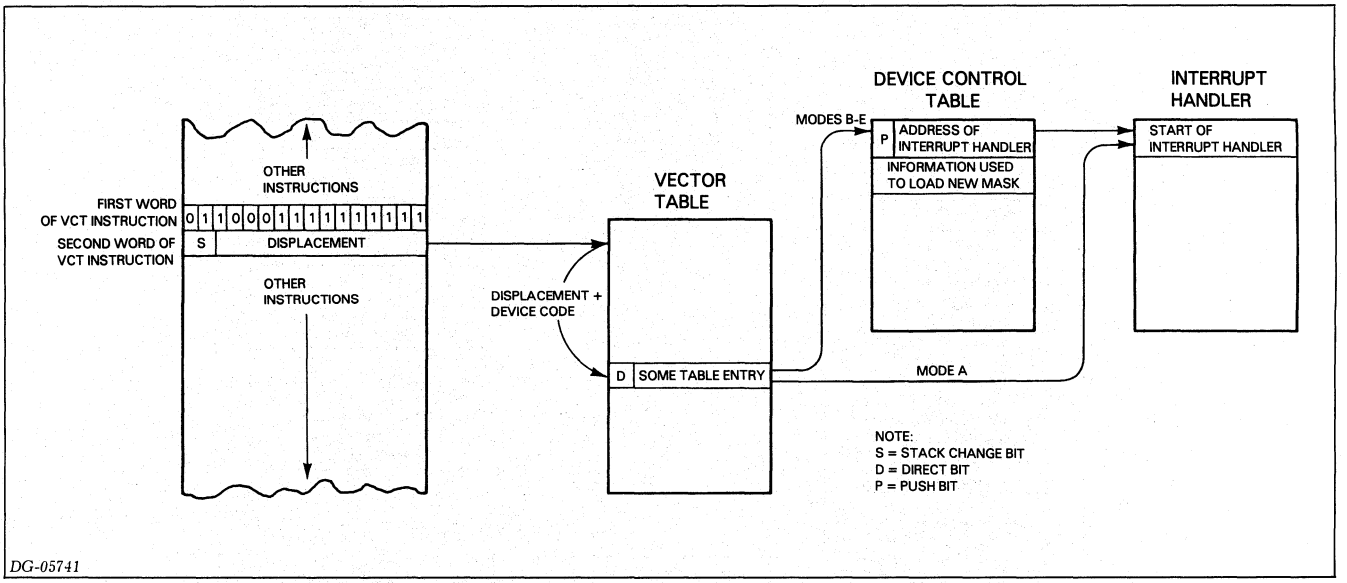
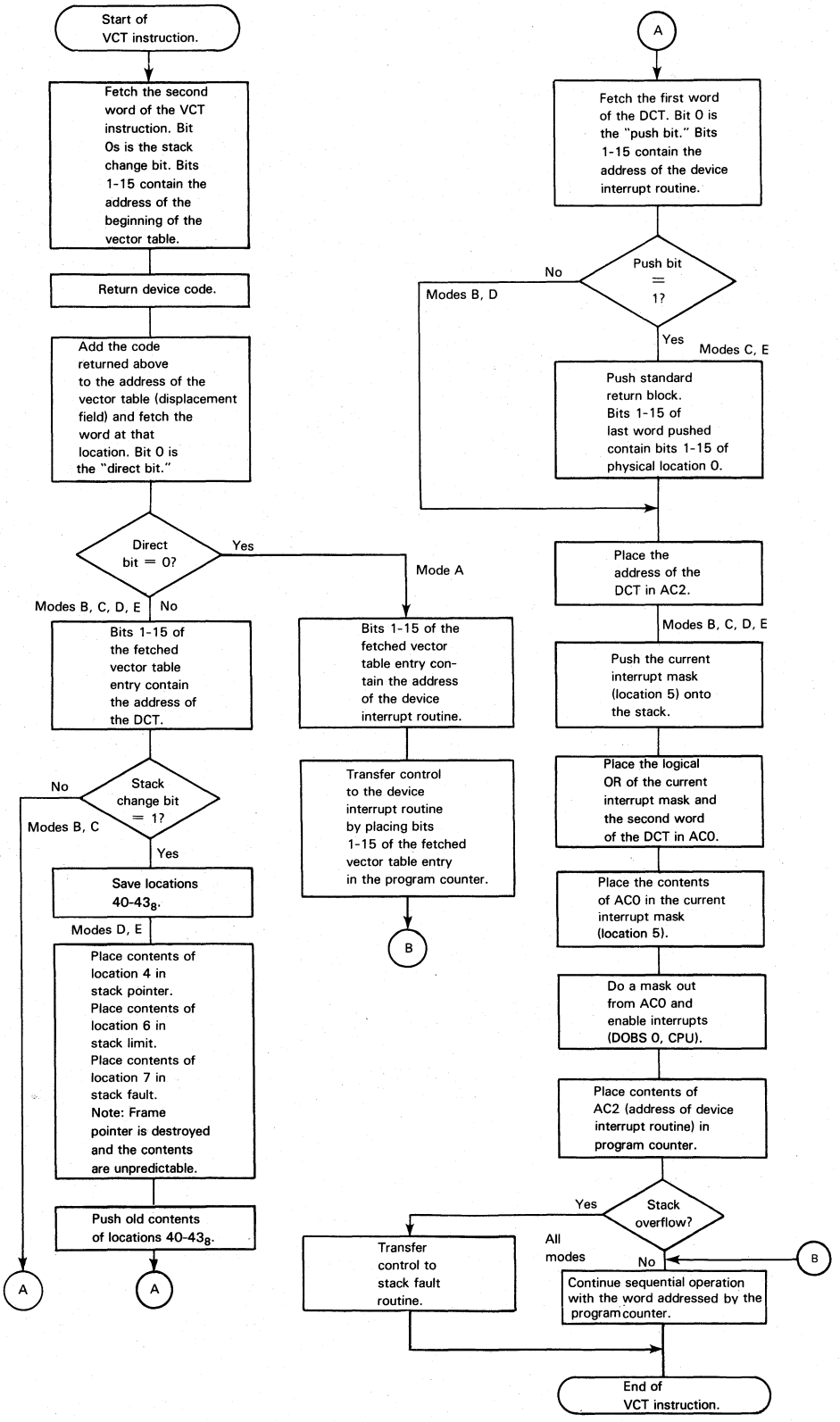


Figure 5.1 Overview of the *Vector* instruction



DG-057

Figure 5.2 Operation of the *Vector* instruction

In **Mode B** both the stack change and the push bits are 0. The instruction uses the vector table entry as the address of the device control table *(DCT)* for the interrupting device. Bit 1-15 of the first word of the DCT contain the address of the desired interrupt handler (bit 0 is the push bit). The second word of the DCT contains information used to construct the new interrupt priority mask. Succeeding words (if any) contain information to be used by the device interrupt handler.

In **Mode C** the stack change bit is 0 and the push bit is 1. This mode performs the functions of mode B and pushes a standard five-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).

In **Mode D**, the stack change bit is 1 and the push bit is 0. This mode performs the functions of mode B, sets up a new stack for the interrupt handler (using the contents of locations 4, 6, and 7), and pushes the previous contents of physical locations $40\text{-}43_8$ (the user stack control words) onto the new stack.

In **Mode E**, the stack change bit and the push bit are 1. This mode combines the functions of modes C and D. That is, it performs the functions of mode B, sets up a new stack, and pushes a five-word return block and the previous stack control words onto the new stack.

Modes B through E use the same procedure for the remainder of the *Vector* instruction. The instruction pushes the current priority mask (location 5) onto the stack, updates location 5, and performs a *Mask Out* instruction (using the logical OR of the current mask and the second word of the DCT). The instruction then sets the Interrupt On flag to 1 and passes control to the selected device interrupt handler. Note that the CPU permits one more instruction to execute (in this case, the first instruction of the interrupt handler) before the next I/O interrupt can occur.

Burst Multiplexor Channel

Device Code - 5_8 (Primary)

Priority Mask Bit - None

Device Flag Commands

f=**S** Sets the Busy flag to 1 and initiates a BMC map load or dump operation.

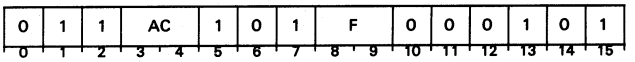
f=**C** Sets the status register (except bits 1 and 15) to 0.

f=**P** No effect.

IORST Sets the status register (except bits 1 and 15) to 0.

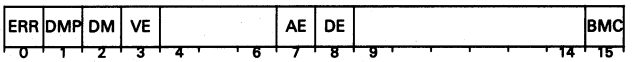
Read BMC Status

DIC*[f]* *ac*,BMC



Reads the BMC status.

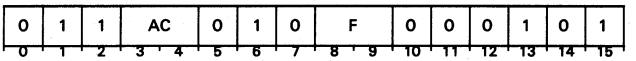
Places the contents of the BMC status register into the specified accumulator. The previous contents of the accumulator are lost. After the data transfer, performs the function specified by f. The format of the specified accumulator is as follows:



Bits	Name	Contents or Function
0	Error	If 1, the BMC detected a validity protect error, an address parity error, or a data parity error.
1	Dump	If 1, the next map transfer operation will be a map dump. If 0, the next map transfer operation will be a map load.
2	Diagnostic Mode	If 1, the BMC is in two-step diagnostic mode.
3	Validity Error	If 1, the BMC detected a validity protect error.
4-6	—	Reserved for future use.
7	Address Error	If 1, the BMC detected an address parity error.
8	Data Error	If 1, the BMC detected a data parity error.
9-14	—	Reserved for future use.
15	BMC	If 1, the BMC is present in the system.

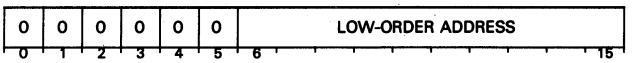
Specify Low-Order Address

DOA*[f]* *ac*,BMC



Specifies the 10 low-order address bits of the first memory location to be loaded or dumped during the next map transfer operation.

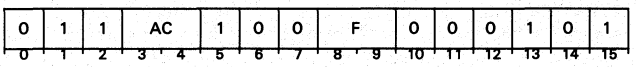
Places bits 6-15 of the specified accumulator into the map transfer address register. The contents of the accumulator remain unchanged. After the data transfer, performs the function specified by f. The format of the specified accumulator is as follows:



Bits	Name	Contents or Function
0-5	—	Must be 0.
6-15	Low-order Address	The 10 low-order bits of the 20-bit physical address of the first memory location to supply or receive a word to/from the specified map register during the next map transfer operation.

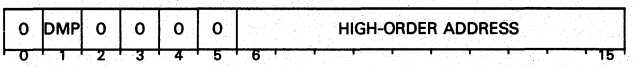
Specify Operation and High-Order Address

DOB*[f]* *ac*,BMC



Specifies the next map transfer operation (load or dump) and the 10 high-order address bits of the first memory location to supply or receive a word during the operation.

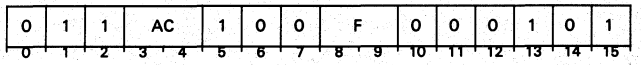
Places bits 1 and 6-15 of the specified accumulator into the map transfer address register. The contents of the accumulator remain unchanged. After the data transfer, performs the function specified by f. The contents of the accumulator are as follows:



Bits	Name	Contents or Function
0	—	Must be 0.
1	Dump	If 1, specifies a dump map as the next map transfer operation. If 0, specifies a load map as the next map transfer operation.
2-5	—	Must be 0.
6-15	High-order	The 10 high-order bits of the 20-bit physical address of the first memory location to supply or receive a word to/from the specified map register during the next map transfer operation.

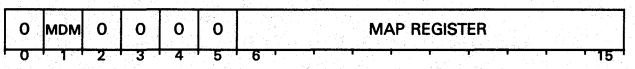
Specify Initial Map Register

DOB*[f]* *ac*,BMC



Specifies the first map register to receive or supply a word during the next map transfer (load or dump) operation.

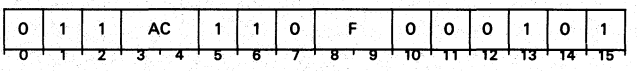
Places bits 1 and 6-15 of the specified accumulator into the map register selector. The contents of the accumulator remain unchanged. After the data transfer, performs the function specified by f. The contents of the accumulator are as follows:



Bits	Name	Contents or Function
0	—	Must be 1.
1	Mapped Diagnostic Mode	If 1, enables mapping in two-step diagnostic mode only.
2-5	—	Must be 0.
6-15	Map Register	The 10-bit number specifying the first map register to either receive or supply a word to/from memory during the next map transfer operation. The 5 high-order bits (bits 6-10) specify the map table containing the register and the 5 low-order bits (bits 11-15) specify the logical page number that indexes the desired map register in the map table.

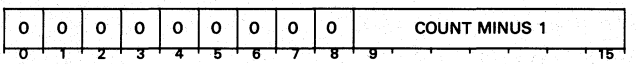
Specify Map Register Count

DOC*[f]* *ac*,BMC



Specifies the number of map registers to be loaded or dumped during the next map transfer operation.

Places bits 9-15 of the specified accumulator into the map register counter. The contents of the accumulator remain unchanged. After the data transfer, performs the function specified by f. The contents of the accumulator are as follows:



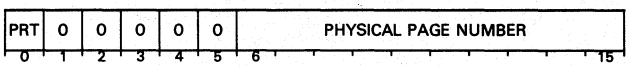
Bits	Name	Contents or Function
0-8	—	Must be 0.
9-15	Map Count	Specifies a number that is one less than the number of map registers to be loaded or dumped during the next map transfer operation.

Map Transfer Operations

In all BMC map transfers, the map register selector specifies the map register involved and the map transfer address register specifies the memory location. During a load map operation, the BMC places the 16-bit contents of the memory location into the map register. During a dump map operation, the BMC places the 16-bit contents of the map register into the memory location.

After the data transfer in each operation, the map register selector and the map transfer address register are incremented by one to select the next consecutive map register and memory location, respectively, and the map register counter is decremented by one. The BMC continues to transfer data between consecutive map registers and memory locations until the contents of the map counter equals 0. After the last map register in the selected map table is accessed, the next consecutive map table is selected. Thus, a total of four consecutive map tables can be transferred during one map transfer operation.

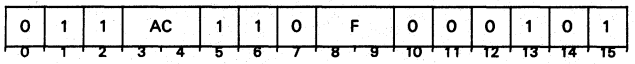
The format of the data in the addressed memory location or map register is as follows:



Bits	Name	Contents or Function
0	Protect	If 1, the BMC cannot transfer data to or from memory locations in the specified physical page.
1-5	—	Must be 0.
6-15	Physical Page	The 10-bit physical page number.

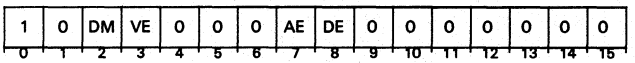
Load BMC Status

DOC*[f]* *ac*,BMC



Defines the diagnostic functions of the BMC.

Places the contents of the specified accumulator into the BMC status register. The contents of the accumulator remain unchanged. After the data transfer, performs the function specified by f. The contents of the accumulator are as follows:



Bits	Name	Contents or Function
0	—	Must be 1.
1	—	Must be 0.
2	DM	If 1, the BMC enters two-step diagnostic mode.
3	VE	If 1, the BMC forces a validity protect error.
4-6	—	Must be 0.
7	AE	If 1, the BMC forces an address parity error.
8	DE	If 1, the BMC forces a data parity error.
9-15	—	Must be 0.

ERCC Error Correction

Device Code - 2_8

Priority Mask Bit - None

Device Flag Commands

f=**S** Sets the interrupt request flag and the Done flag to 0.

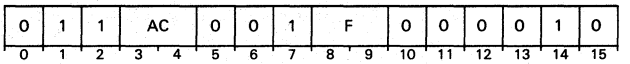
f=**C** No effect.

f=**P** No effect.

IORST Sets ERCC to 010 state; clears Done flag.

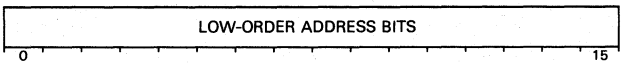
Read Memory Fault Address

DIA*[f]* *ac*,ERCC



Places the sixteen low-order bits of the physical address of the fault location into the specified accumulator. (The previous contents of that accumulator are lost.) The instruction sets the Done flag as specified by the flag command.

The following shows the format of the contents of the specified accumulator.

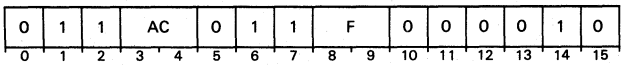


Bits	Name	Contents or Function
0-15	Low-order Address	Sixteen low-order bits of the physical address of the memory fault location.

NOTE: *The physical address is meaningless unless it is read after the ERCC facility requests an interrupt and before a* Start *or* **IORST** *flag command sets the Done flag to 0.*

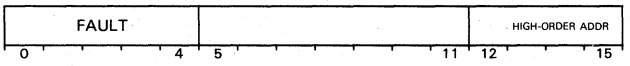
Read Memory Fault Code

DIB*[f]* *ac*,ERCC



Places a 5-bit error identification code in bits 0-4 of the specified accumulator. The instruction first sets bits 5-11 of the accumulator to 0 and places the four high-order bits of the physical address of the fault location in bits 12-15. Next, the instruction sets the Done flag as specified by the flag command.

The following table shows the format of the contents of the specified accumulator.

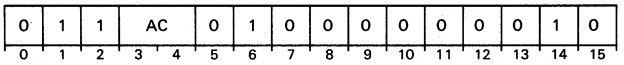


Bits	Name	Contents or Function
0-4	Fault Code	5-bit code, identifies the bit in error. 00000 No error. 00001 Check bit 4. 00010 Check bit 3. 00011 Data bit 0. 00100 Check bit 2. 00101 Data bit 1. 00110 Multiple bit error. 00111 Data bit 3. 01000 Check bit 1. 01001 Data bit 4. 01010 All 21 bits in memory are 1. 01011 Data bit 6. 01100 Data bit 7. 01101 Data bit 8. 01110 Data bit 9. 01111 Multiple bit error. 10000 Check bit 0. 10001 Data bit 11. 10010 Data bit 12. 10011 Data bit 13. 10100 Data bit 14. 10101 All 21 bits in memory are 0. 10110 Data bit 2. 10111 Multiple bit error. 11000 Data bit 10. 11001 Multiple bit error. 11010 Data bit 5. 11011 Multiple bit error. 11100 Data bit 15. 11101 Multiple bit error. 11110 Multiple bit error. 11111 Multiple bit error.
5-11	—	Reserved for future use.
12-15	High Order Address	Four high-order bits of the physical address of the fault location.

NOTE: *The address is meaningless unless read after the ERCC facility requests an interrupt and before a* Start *or* **IORST** *flag command sets the Done flag to 0.*

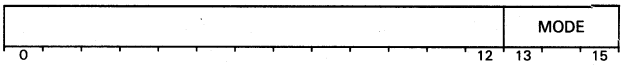
Enable ERCC

DOA*[f]* *ac*,ERCC



Sets the ERCC facility to function according to bits 13-15 of the specified accumulator. Next, the instruction sets the Done flag and then the Interrupt Request flag, as specified by the flag command. The instruction disregards bits 0-12.

The following shows the format of the contents of the specified accumulator. Modes designated with an asterisk (*) were not implemented in previous ERCC instruction sets.



Bits	Name	Contents or Function
0-12	—	Reserved for future use.
13-15	MODE	Control the ERCC feature as follows: 000 or 001* Write checkword; disable checking and correction disable interrupts. 010 Write checkword; check data and checkword; send correct data to processor; disable interrupts. 011 Write checkword; check data and checkword; send correct data to processor; enable interrupts. 100* or 101* Duplicate five low-order data bits in checkword; disable checking and correcting; disable interrupts. 110* Duplicate five low-order data bits in checkword; compare five low-order bits to checkword; send data with checkword to processor; disable interrupts. 111* Write five low-order data bits as checkword; compare five low-order bits to checkword; send data with checkword to processor; interrupt on error.

Memory Management and Protection Unit

Device Code - 3_8 (Primary)

Priority Mask Bit - None

Device Flag Commands

f=**S** No effect.

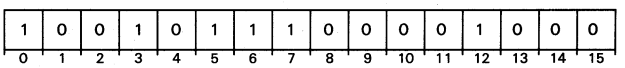
f=**C** No effect.

f=**P** Enables Map Single Cycle.

IORST Disables MMPU; clears Map Status Register.

Load Map

LMP



Under control of AC1 and AC2, loads successive words from memory into the MMPU where they are used to define a user or data channel map.

AC1 must contain an unsigned integer, which is the number of words to be loaded into the MMPU. Bits 1-15 of AC2 must contain the address of the first word to be loaded. If bit 0 of AC2 is 1, the instruction follows the indirection chain and places the resultant effective address in AC2. AC0 and AC3 are ignored and their contents remain unchanged.

For each word loaded, the instruction decrements the count in AC1 by one and increments the source address in AC2 by 1. Upon completion of the instruction, AC1 contains 0, and AC2 contains the address of the word following the last word loaded.

This instruction is interruptible in the same manner as the *Block Add and Move* instruction.

The words loaded into the MMPU define the address translation functions for the various user and data channel maps. The contents of the **MAP** field (bits 6-8) of the MMPU status register determine which map is affected by the *Load Map* instruction. You can alter this field using either the *Load Map Status* or the *Initiate Page Check* instruction.

The format of the words loaded into the MMPU is as follows:

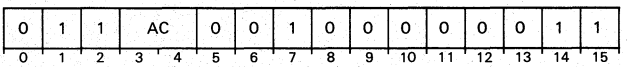
Bits	Name	Contents or Function
0	WRITE PROTECT	Must be 0 for data channel maps; 1 for for user maps.
1-5	LOGICAL	Logical page number.
6-15	PHYSICAL	Physical page number.

NOTE: *Declare a logical page invalid by setting the write protect bit to 1 and all of bits 6-15 to 1.*

If I/O protection is enabled, execution of the Load Map *instruction will cause a trap.*

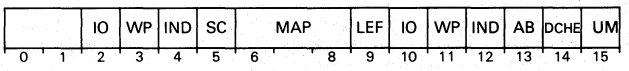
Read Map Status

DIA*[f]* *ac*,**MAP**



Reads the status of the current map.

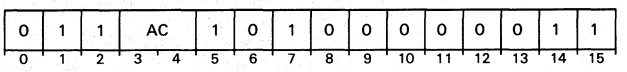
Places the contents of the MMPU status register in the specified AC. The previous contents of the specified AC are lost. The format of the information placed in the specified AC is as follows:



Bits	Name	Contents or Function
0-1	—	Reserved for future use.
2	I/O	If 1, the last protection fault was an I/O protection fault.
3	WP	If 1, the last protection fault was a write or validity protection fault.
4	IND	If 1, the last protection fault was an indirect protection fault.
5	Single Cycle	If 1, the last map fault occurred during a single cycle memory reference.
6-8	Map	Indicates which map will be loaded by the next *Load map* instruction: 000 User A 001 Reserved for future use. 010 User B 011 Reserved for future use. 100 Data channel A 101 Data channel C 110 Data channel B 111 Data channel D
9	LEF	If 1, the *Load Effective Address* instruction was enabled by the last *Load Map Status* instruction.
10	I/O	If 1, I/O protection was enabled by the last *Load Map Status* instruction.
11	WP	If 1, write protection was enabled by the last *Load Map Status* instruction.
12	IND	If 1, indirect protection was enabled by the last *Load Map Status* instruction.
13	A/B	If 0, the last *Load Map Status* instruction enabled user map A. If 1, the last *Load Map Status* instruction enabled user map B.
14	DCH Enable	If 1, the mapping of the data channel addresses is enabled.
15	User Mode	If 1, the last I/O interrupt occurred while in user mode.

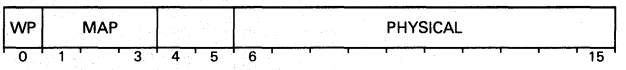
Page Check

DIC *ac*,MAP



Identifies and provides some characteristics of the physical page corresponding to a logical page. The logical page was identified by an *Initiate Page Check* instruction.

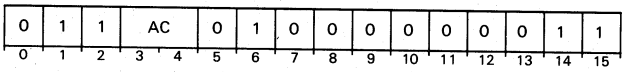
Places the number of the physical page in bits 6-15 of the specified AC, places other information about the page in bits 0-3, and destroys the previous contents of the AC. The format of the information placed in the specified AC is as follows:



Bits	Name	Contents or Function
0	WP	The write protect bit for the logical page which corresponds to the physical page specified by bits 6-15.
1-3	Map	The map used to perform the translation between logical page number and physical page number: 000 User A 001 Reserved. 010 User B 011 Reserved. 100 Data channel A 101 Data channel C 110 Data channel B 111 Data channel D
4-5	—	Reserved for future use.
6-15	Physical	The number of the physical page which corresponds to the logical page given in the preceding *Initiate Page Check* instruction. If all these bits are 1, and WP (bit 0) is 1, then the logical page is validity protected.

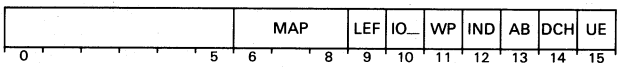
Load Map Status

DOA *ac*,MAP



Defines the parameters of a new map.

Places the contents of the specified AC in the MMPU status register. The contents of the specified AC remain unchanged. The format of the specified AC is as follows:

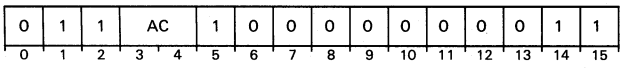


Bits	Name	Contents or Function
0-5	—	Reserved for future use.
6-8	Map Select	Specify which map will be loaded by the next Load Map instruction: 000 User A 001 Reserved for future use. 010 User B 011 Reserved for future use. 100 Data channel A 101 Data channel C 110 Data channel B 111 Data channel D
9	Lef	If 1, the Load Effective Address instruction will be enabled for the next user.
10	I/O	If 1, I/O protection will be enabled for the next user.
11	WP	If 1, write protection will be enabled for the next user.
12	IND	If 1, indirect protection will be enabled for the next user.
13	A/B	If 0, user map A will be enabled next. If 1, user map B will be enabled next.
14	DCH Enable	If 1, the mapping of data channel addresses will be enabled immediately after this instruction.
15	User Enable	If 1, mapping of CPU addresses will commence with the first memory reference *after* the next *indirect* reference or return type instruction (**POPB, POPJ, RTN, RSTR**).

NOTE: *If the* Load Map Status *instruction sets the User Enable bit to 1, this inhibits the interrupt system and the MMPU waits for either an indirect reference (except* **DSPA, BLM, BAM LMP** *floating point, bit or character instruction) or return type instruction. Either event releases the interrupt system and allows the MMPU to begin translating addresses (using the user map specified by bit 13 of the MMPU status register). Address translation resumes (1) after the first level of the next indirect reference; or (2) after the first* Pop Block, Pop Jump, Return, *or* Restore *instruction that does not cause a stack fault.*

Map Page 31

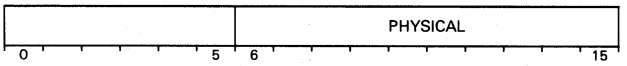
DOB *ac*,MAP



Specifies that mapping take place for a single page of an unmapped address space. Mapping is always done for locations 76000_8 through 77777_8 (logical page 31). This is the only page which can be mapped when in unmapped address space. You can use this instruction to access a page of a user's memory space when in unmapped mode.

Bits 6-15 of the specified AC are transferred to the MMPU. These bits specify a physical page number to which logical page 31 will be mapped when in the unmapped mode.

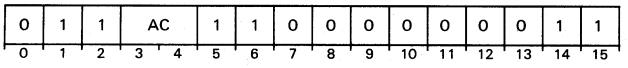
The contents of the specified AC remain unchanged. The format of the specified AC is as follows:



Bits	Name	Contents or Function
0-5	—	Reserved for future use.
6-15	Physical	The number of the physical page to which logical page 31 should be mapped when in unmapped mode.

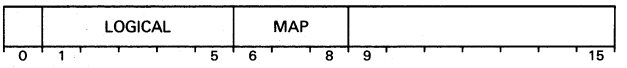
Initiate Page Check

DOC *ac*,MAP



Identifies a logical page. The *Page Check* instruction will find the corresponding physical page.

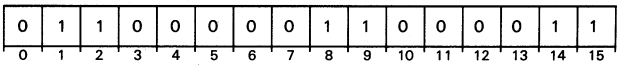
Transfers the contents of the specified AC to the MMPU for later use by the *Page Check* or *Load Map* instruction. Leaves the contents of the specified AC unchanged. The format of the specified AC is as follows:



Bits	Name	Contents or Function
0	—	Reserved for future use.
1-5	Logical Page	Number of the logical block for which the check is requested.
6-8	Map	Specify which map should be used for the check as follows: 000 User A. 001 Reserved. 010 User B. 011 Reserved. 100 Data channel A. 101 Data channel C. 110 Data channel B. 111 Data channel D.
9-15	—	Reserved for future use.

Map Single Cycle Disable User Mode

NIOP *ac*,MAP



Issued from unmapped mode, the instruction maps one memory reference using the last user map; issued from User mode with *Lef* mode and I/O protection disabled, the instruction simply turns off the map, returning it to unmapped mode. It is used by the supervisor to access a user's memory space when only one or two references are required. It is also used by a privileged user to turn off memory mapping.

From unmapped mode — Enables the user map for one memory reference. Maps the first memory reference of the next **LDA** or **STA** instruction. After the memory cycle is mapped, the instruction again disables the user map.

The interrupt system is disabled from the beginning of the *Map Single Cycle* instruction until after the next **LDA** or **STA** instruction.

From user mode — If *LEF* mode and I/O protection are disabled, this instruction turns off the MMPU. All subsequent memory references are unmapped until a map is reactivated with a *Load Map Status* instruction.

This instruction, when issued to logical page 31, gives undefined results. Single cycle memory references to validity protected pages also give undefined results. No validity trap is generated.

Real Time Clock

Device Code - 14_8 (Primary)

Priority Mask Bit - 13

Device Flag Commands

f=**S** Sets the Busy flag to 1, and the Done flag and interrupt request flag to 0; enables RTC interrupts.

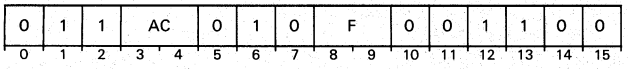
f=**C** Sets the Busy and Done flags and the interrupt request flag to 0; disables RTC interrupts.

f=**P** No effect.

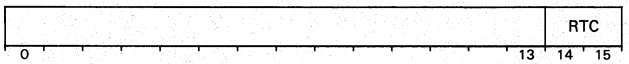
IORST Sets the Busy and Done flags, the interrupt request flag, the interrupt mask bit (bit 13), and the clock frequency select bits to 0; disables RTC interrupts.

Select RTC Frequency

DOA*[f]* *ac*,**RTC**



The clock frequency is set according to bits 14-15 of the specified AC. The contents of the specified AC remain unchanged. Bits 0-13 of the specified AC are ignored. The format of the specified AC is as follows:



Bits	Name	Contents or Function
0-13	—	Reserved for future use. (Set to 0)
14-15	RTC	Selects the clock frequency as follows: 00 AC line frequency 01 10Hz 10 100Hz 11 1000Hz

Primary Asynchronous Line Input

Device Code - 10_8 (Primary)

Priority Mask Bit - 14

Device Flag Commands

f=**S** Sets the Busy flag to 1 and the Done flag to 0.

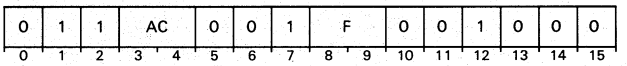
f=**C** Sets the Busy and Done flags to 0.

f=**P** No effect.

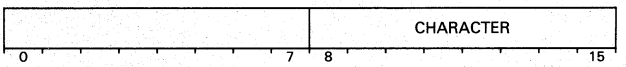
IORST Sets the Busy and Done flags to 0.

Read Character Buffer

DIA*[f]* *ac*,**TTI**



Places the contents of the controller's input buffer in bits 8-15 of the specified accumulator. After the data transfer, sets the controller's Busy and Done flags according to the function specified by **F**. The format of the specified accumulator is as follows:



Bits	Name	Contents or Function
0-7	—	Reserved for future use.
8-15	Character	The character read from the input buffer, right-justified.

Primary Asynchronous Line Output

Device Code - 11_8 (Primary)

Priority Mask Bit - 15

Device Flag Commands

f=**S** Sets the Busy flag to 1 and the Done flag to 0; begins transmission of the character contained in the output buffer.

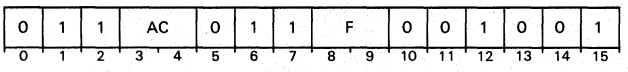
f=**C** Sets the Busy and Done flags and the interrupt request flag to 0.

f=**P** No effect.

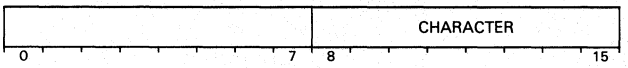
IORST Sets the Busy and Done flags, the interrupt request flag, and the interrupt mask bit (bit 15) to 0.

Load Character Buffer

DOA*[f]* *ac*,TTO

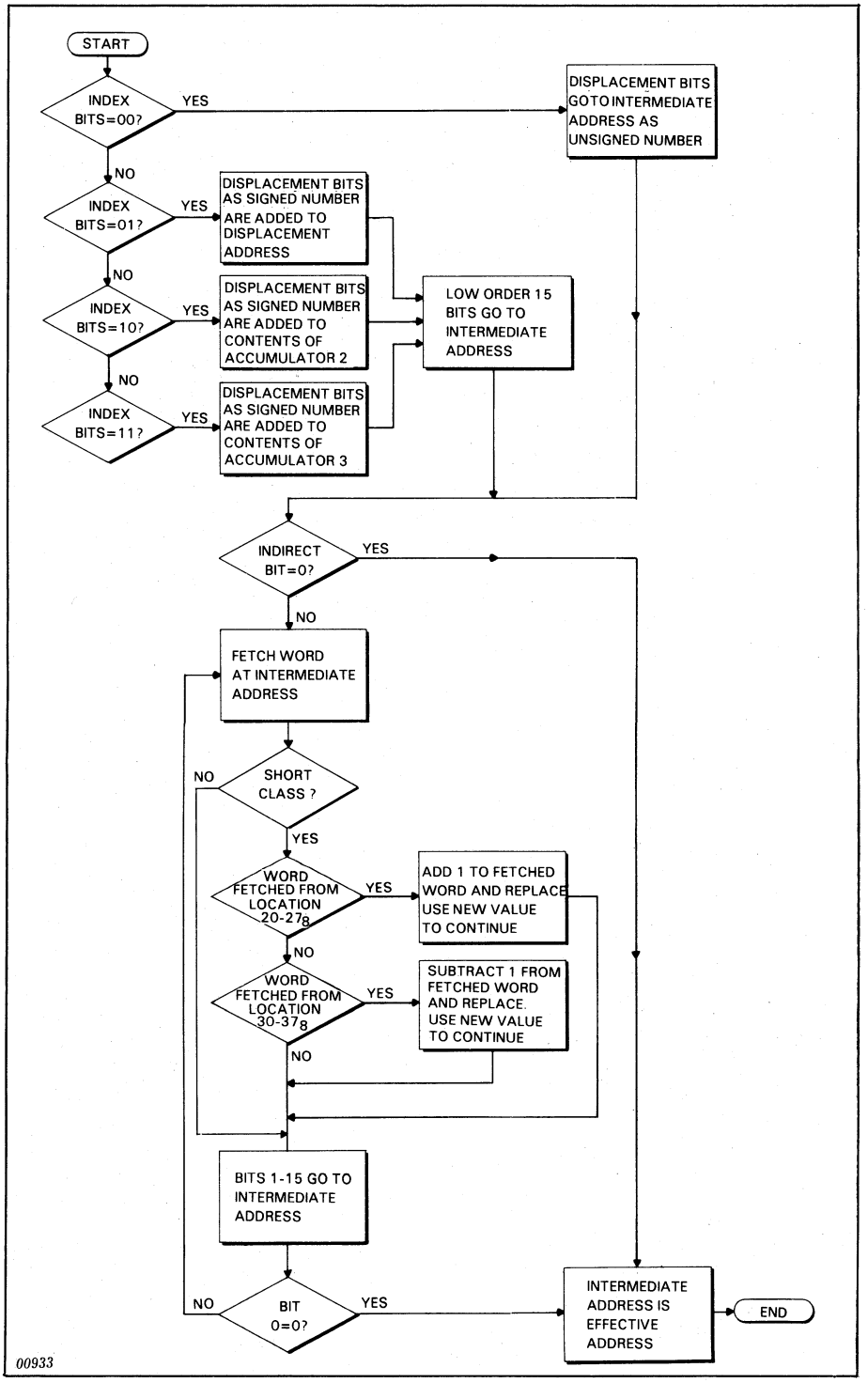


Loads bits 8-15 of the specified accumulator into the controller's output buffer. After the data transfer, sets the controller's Busy and Done flags according to the function specified by **F**. The contents of the specified accumulator remain unchanged. The format of the specified accumulator is as follows:



Bits	Name	Contents or Function
0-7	—	Reserved for future use.
8-15	Character	The character, right-justified, to be placed in the output buffer.

Appendix A
The Addressing Process



Appendix B
Standard I/O Device Codes

Octal Device Code	Mnem	Priority Mask Bit	Device Name
00	—	—	Unused
01	WCS	—	Writeable control store option [or APL
	[APL]	—	register]
02	ERCC	—	Error checking and correction
	MAP		Memory allocation and protection
03			
04			
05	BMC	—	Burst multiplexor channel
06	MCAT	12	Multiprocessor adapter transmitter
			Multiprocessor adapter receiver
07	MCAR	12	TTY input
			TTY output
10	TTI	14	
11	TTO	15	
12	PTR	11	Paper tape reader
13	PTP	13	Paper tape punch
14	RTC	13	Real-time clock
15	PLT	12	Incremental plotter
16	CDR	10	Card reader
17	LPT	12	Line printer
20	DSK	9	Fixed-head disc
21	ADCV	8	A-D converter
22	MTA	10	Magnetic tape
23	DACV	None	D-A converter
24	DCM	0	Data communications multiplexor
25			Fixed-head DG/Disc
26	DKB	9	DG/Disk storage subsystem
27	DPF	7	Asynchronous hardware multiplexor
30	QTY	14	
30	SLA	14	Synchronous line adapter
31[1]	IBM1	13	IBM 360/370 interface
32	IBM2	13	IBM 360/370 interface
33	DKP	7	Moving head disk
34	CAS[1]	10	Cassette tape
	DCU[4]	4	Data control unit
34	MX1	11	Multiline asynchronous controller
35	MX2	11	Multiline asynchronous controller
36	IPB	6	Interprocessor bus—half-duplex
37	IVT	6	IPB watchdog timer
40[2]	DPI	8	IPB full-duplex input
40	SCR	8	Synchronous communication receiver

Octal Device Code	Mnem	Priority Mask Bit	Device Name
41[3]	DPO	8	IPB full-duplex output
41	SCT	8	Synchronous communication transmitter
			Digital I/O
42	DIO	7	Digital I/O timer
43	DIOT	6	Programmable interval timer
43	PIT	11	
44	MXM	12	Modem control for MX1/MX2
45			Second multiprocessor transmitter
46	MCAT1	12	Second multiprocessor receiver
			Second TTY input
47	MCAR1	12	
50	TTI1	14	
51	TTO1	15	Second TTY output
52	PTR1	11	Second paper tape reader
53	PTP1	13	Second paper tape punch
54	RTC1	13	Second real-time clock
55	PLT1	12	Second incremental plotter
56	CDR1	10	Second card reader
57	LPT1	12	Second line printer
60	DSK1	9	Second fixed-head disk
61	ADCV1	8	Second A-D converter
62	MTA1	10	Second magnetic tape
63	DACV1	None	Second D-A converter
64			
65	IOP1	5[5]	Host to IOP interface
66	DKB1	9	Second fixed-head DG/Disk
67	DPF1	7	Second DG/Disk storage subsystem
70	QTY1	14	Second asynchronous hardware multi-
			plexor
70	SLA1	14	Second synchronous line adapter
			Second IBM 360/370 interface
71[1]			Second IBM 360/370 interface
		13	Second moving head disk
72			
	DKP1	7	
73			
74	CAS1	10	Second cassette tape
74[1]		11	Second multiline asynchronous controller
75		11	Second multiline asynchronous controller
76	DPU	4	DCU to host interface
77	CPU	—	Central processor and console functions

[1]*Code returned by INTA and used by VCT.*

[2]*Can be set up with any unused even device code equal to 40 or above.*

[3]*Can be set up with any unused odd device code equal to 41 or above.*

[4]*Can be set to any unused device code between 1 and 76.*

[5]*Micro interrupts are not maskable.*

Appendix C
The ASCII Character Codes

DECIMAL	OCTAL	HEX	KEY SYMBOL	MNEMONIC
0	000	00	↑@	NUL
1	001	01	↑A	SOH
2	002	02	↑B	STX
3	003	03	↑C	ETX
4	004	04	↑D	EOT
5	005	05	↑E	ENQ
6	006	06	↑F	ACK
7	007	07	↑G	BEL
8	010	08	↑H	BS (BACKSPACE)
9	011	09	↑I	TAB
10	012	0A	↑J	NEW LINE
11	013	0B	↑K	VT (VERT.TAB)
12	014	0C	↑L	FORM FEED
13	015	0D	↑M	CARRIAGE RETURN
14	016	0E	↑N	SO
15	017	0F	↑O	SI
16	020	10	↑P	DLE
17	021	11	↑Q	DC1
18	022	12	↑R	DC2
19	023	13	↑S	DC3
20	024	14	↑T	DC4
21	025	15	↑U	NAK
22	026	16	↑V	SYN
23	027	17	↑W	ETB
24	030	18	↑X	CAN
25	031	19	↑Y	EM
26	032	1A	↑Z	SUB
27	033	1B	ESC	ESCAPE
28	034	1C	↑\	FS
29	035	1D	↑]	GS
30	036	1E	↑↑	RS
31	037	1F	↑ –	US

DECIMAL	OCTAL	HEX	KEY SYMBOL
32	040	20	SPACE
33	041	21	!
34	042	22	" (QUOTE)
35	043	23	#
36	044	24	$
37	045	25	%
38	046	26	&
39	047	27	' (APOS)
40	050	28	(
41	051	29	)
42	052	2A	*
43	053	2B	+
44	054	2C	, (COMMA)
45	055	2D	-
46	056	2E	. (PERIOD)
47	057	2F	/
48	060	30	0
49	061	31	1
50	062	32	2
51	063	33	3
52	064	34	4
53	065	35	5
54	066	36	6
55	067	37	7
56	070	38	8
57	071	39	9
58	072	3A	:
59	073	3B	;
60	074	3C	<
61	075	3D	=
62	076	3E	>
63	077	3F	?
64	100	40	@

DECIMAL	OCTAL	HEX	KEY SYMBOL
65	101	41	A
66	102	42	B
67	103	43	C
68	104	44	D
69	105	45	E
70	106	46	F
71	107	47	G
72	110	48	H
73	111	49	I
74	112	4A	J
75	113	4B	K
76	114	4C	L
77	115	4D	M
78	116	4E	N
79	117	4F	O
80	120	50	P
81	121	51	Q
82	122	52	R
83	123	53	S
84	124	54	T
85	125	55	U
86	126	56	V
87	127	57	W
88	130	58	X
89	131	59	Y
90	132	5A	Z
91	133	5B	[
92	134	5C	\
93	135	5D	]
94	136	5E	↑ OR ^
95	137	5F	← OR _
96	140	60	` (GRAVE)

DECIMAL	OCTAL	HEX	KEY SYMBOL
97	141	61	a
98	142	62	b
99	143	63	c
100	144	64	d
101	145	65	e
102	146	66	f
103	147	67	g
104	150	68	h
105	151	69	i
106	152	6A	j
107	153	6B	k
108	154	6C	l
109	155	6D	m
110	156	6E	n
111	157	6F	o
112	160	70	p
113	161	71	q
114	162	72	r
115	163	73	s
116	164	74	t
117	165	75	u
118	166	76	v
119	167	77	w
120	170	78	x
121	171	79	y
122	172	7A	z
123	173	7B	{
124	174	7C	|
125	175	7D	}
126	176	7E	~ (TILDE)
127	177	7F	DEL (RUBOUT)

DG-05495

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: Bogato
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

DataGeneral

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

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 ☐Analyst/Programmer ☐Other _____
☐Senior Systems Analyst ☐Operator _____
☐Engineer ☐End User

How do you use this manual? *(List in order: 1 = Primary Use)*

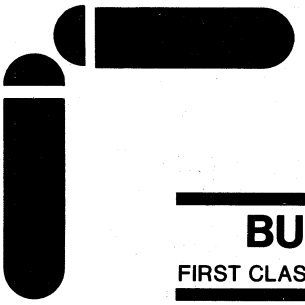
___ Introduction to the product ___ Tutorial Text ___ Other
___ Reference ___ Operating Guide _____

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:



NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 26 SOUTHBORO, MA 01772

Postage will be paid by addressee

DataGeneral

Customer Documentation
MS E-219
4400 Computer Drive
Westboro, MA 01581-9973

Data General Corporation, Westboro, MA 01580

014-000642-04