Programmer's Reference Series

ECLIPSE® Line Computers

Data General

Programmer's Reference Series

ECLIPSE® Line Computers

DataGeneral

NOTICE

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

Users are cautioned that DGC reserves the right to make changes without notice in the specifications and materials contained herein and shall not be responsible for any damages (including consequential) caused by reliance on the materials presented, including, but not limited to typographical, arithmetic, or listing errors.

Ordering No. 014-000626

Rev. 02, October 1980

CONTENTS

CHAPTER I THE ECLIPSE COMPUTER

1 INTRODUCTION

1 FEATURES OF THE ECLIPSE LINE OF COMPUTERS
1 Powerful Basic Instruction Set
1 Stack
1 Floating Point
2 Commercial Instruction Set
2 Memory Allocation and Protection
2 Extended Operation
3 Writeable Control Store
3 Error Checking and Correction
3 Memory Features
4 Power Fail/Auto-restart
4 Real-time Clock
4 Input/Output Bus
5 Input/Output Devices
5 Software
5 Conclusion

CHAPTER II INTERNAL STRUCTURE

1 INTRODUCTION

1 INFORMATION FORMATS
1 Bit Numbering
2 Octal Representation
2 Character Codes
2 Information Representation

5 INFORMATION ADDRESSING
5 Word Addressing
7 Byte Addressing
8 Bit Addressing
9 Addressing With The MAP Feature

10 RESERVED STORAGE LOCATIONS

11 PROGRAM EXECUTION
11 Program Flow Alteration
12 Program Flow Interruption

CHAPTER III INSTRUCTION SETS

1 INTRODUCTION

1 INSTRUCTION FORMATS

2 CODING AIDS

4 FIXED POINT ARITHMETIC
4 LOAD ACCUMULATOR
4 STORE ACCUMULATOR
4 ADD
4 SUBTRACT
4 DECIMAL ADD
5 DECIMAL SUBTRACT
5 ADD IMMEDIATE
5 EXTENDED ADD IMMEDIATE
6 SUBTRACT IMMEDIATE
6 NEGATE
6 ADD COMPLEMENT
6 MOVE
6 INCREMENT
7 EXCHANGE ACCUMULATORS
7 UNSIGNED MULTIPLY
7 SIGNED MULTIPLY
7 UNSIGNED DIVIDE
7 SIGNED DIVIDE
8 SIGN EXTEND AND DIVIDE
8 HALVE
9 LOGICAL OPERATIONS
9 LOAD EFFECTIVE ADDRESS
9 COMPLEMENT
9 AND
9 AND IMMEDIATE
9 INCLUSIVE OR
9 INCLUSIVE OR IMMEDIATE
9 EXCLUSIVE OR

CONTENTS CONTINUED

10 EXCLUSIVE OR IMMEDIATE
10 AND WITH COMPLEMENTED SOURCE
11 LOGICAL SHIFT
11 DOUBLE LOGICAL SHIFT
11 HEX SHIFT LEFT
11 HEX SHIFT RIGHT
12 DOUBLE HEX SHIFT LEFT
12 DOUBLE HEX SHIFT RIGHT

13 BYTE MANIPULATION
13 LOAD BYTE
13 STORE BYTE

14 BIT MANIPULATION
14 SET BIT TO ONE
14 SET BIT TO ZERO
14 SKIP ON ZERO BIT
14 SKIP ON NON-ZERO BIT
14 SKIP ON ZERO BIT AND SET TO ONE
15 LOCATE LEAD BIT
15 LOCATE AND RESET LEAD BIT
15 COUNT BITS

16 DATA MOVEMENT
16 BLOCK ADD AND MOVE
17 BLOCK MOVE

18 STACK MANIPULATION
18 Stack Pointer
18 Frame Pointer
18 Stack Limit
18 Stack Fault Address
18 Return Block
19 Stack Frames
19 Stack Protection
19 Stack Protection Faults
20 Initialization of the Stack Control Words

21 STACK MANIPULATION INSTRUCTIONS
21 PUSH MULTIPLE ACCUMULATORS
21 POP MULTIPLE ACCUMULATORS
21 PUSH RETURN ADDRESS
21 SAVE
22 MODIFY STACK POINTER

23 PROGRAM FLOW ALTERATION
23 JUMP
23 JUMP TO SUBROUTINE
23 INCREMENT AND SKIP IF ZERO
24 DECREMENT AND SKIP IF ZERO
24 SKIP IF ACS GREATER THAN ACD
24 SKIP IF ACS GREATER THAN OR EQUAL TO ACD
24 COMPARE TO LIMITS
24 EXECUTE
25 SYSTEM CALL
25 PUSH JUMP
25 POP PC AND JUMP
25 DISPATCH
26 POP BLOCK
26 RETURN
26 RESTORE

27 SUBROUTINES CALLS AND RETURNS

29 EXTENDED OPERATION FEATURE
29 EXTENDED OPERATION

30 WRITEABLE CONTROL STORE FEATURE
30 Placing Microcode in WCS
30 SPECIFY ADDRESS
30 LOAD MICROCODE
30 LOAD DECODE ADDRESS
30 ENTER WCS

31 MEMORY ALLOCATION AND PROTECTION FEATURE
31 Address Translation
31 Sharing of Physical Memory
31 Types of Maps
32 Supervisor Mode
32 Map Protection Capabilities
32 Map Protection Faults
33 Load Effective Address Mode
33 Initial Conditions

34 S/200 AND C/300 MAP INSTRUCTIONS
34 Load Map
35 Load Map Status
35 Read Map Status
36 Initiate Page Check

36 Read User Status
37 Read Map Status
37 Translate Block
38 C/330 MAP INSTRUCTIONS

42 FLOATING POINT ARITHMETIC
42 Floating Point Registers
43 Floating Point Faults

44 FLOATING POINT INSTRUCTIONS
44 LOAD FLOATING POINT SINGLE
44 LOAD FLOATING POINT DOUBLE
44 STORE FLOATING POINT SINGLE
44 STORE FLOATING POINT DOUBLE
44 FLOAT FROM AC
44 FLOAT FROM MEMORY
44 FIX TO AC
45 FIX TO MEMORY
45 MOVE FLOATING POINT
45 ADD SINGLE (FPAC to FPAC)
45 ADD SINGLE (memory to FPA)
45 ADD DOUBLE (FPAC to FPAC)
45 ADD DOUBLE (memory to FPAC)
46 SUBTRACT SINGLE (FPAC from FPAC)
46 SUBTRACT SINGLE (memory from FPAC)
46 SUBTRACT DOUBLE (FPAC from FPAC)
46 SUBTRACT DOUBLE (memory from FPAC)
47 MULTIPLY SINGLE (FPAC by FPAC)
47 MULTIPLY SINGLE (FPAC by memory)
47 MULTIPLY DOUBLE (FPAC by FPAC)
47 MULTIPLY DOUBLE (FPAC by memory)
48 DIVIDE SINGLE (FPAC by FPAC)
48 DIVIDE SINGLE (FPAC by memory)
48 DIVIDE DOUBLE (FPAC by FPAC)
48 DIVIDE DOUBLE (FPAC by memory)

48 NEGATE
48 NORMALIZE
49 ABSOLUTE VALUE
49 READ HIGH WORD
49 SCALE
49 LOAD EXPONENT
49 HALVE
49 COMPARE FLOATING POINT

50 LOAD FLOATING POINT STATUS
50 STORE FLOATING POINT STATUS
50 TRAP ENABLE
50 TRAP DISABLE
50 CLEAR ERRORS
51 PUSH FLOATING POINT STATE
51 POP FLOATING POINT STATE
52 Arithmetic Test
52 NO SKIP
52 SKIP ALWAYS
52 SKIP ON GREATER THAN ZERO
52 SKIP ON LESS THAN ZERO
52 SKIP ON ZERO
52 SKIP ON LESS THAN OR EQUAL TO ZERO
52 SKIP ON GREATER THAN OR EQUAL TO ZERO
52 SKIP ON NON-ZERO
53 Error Test
53 SKIP ON NO MANTISSA OVERFLOW
53 SKIP ON NO UNDERFLOW
53 SKIP ON NO OVERFLOW
53 SKIP ON NO ZERO DIVIDE
53 SKIP ON NO UNDERFLOW AND NO ZERO DIVIDE
53 SKIP ON NO OVERFLOW AND NO ZERO DIVIDE
53 SKIP ON NO UNDERFLOW AND NO OVERFLOW
53 SKIP ON NO ERROR

54 COMMERCIAL INSTRUCTION SET
54 Commercial Faults
54 I/O Interrupts

55 COMMERCIAL INSTRUCTIONS
55 EXTENDED LOAD BYTE
55 EXTENDED STORE BYTE
55 LOAD INTEGER
56 STORE INTEGER
56 LOAD INTEGER EXTENDED
57 STORE INTEGER EXTENDED
57 INTEGERIZE
57 LOAD SIGN
58 CHARACTER MOVE
58 CHARACTER COMPARE

CONTENTS CONTINUED

59 CHARACTER TRANSLATE
60 CHARACTER MOVE UNTIL TRUE
60 EDIT
61 SET T TO ONE
61 SET T TO ZERO
61 SET S TO ONE
61 SET S TO ZERO
61 ADD TO SI
61 ADD TO DI
62 ADD TO P
62 ADD TO P DEPENDING ON T
62 ADD TO P DEPENDING ON S
62 STORE IN STACK
62 DECREMENT AND JUMP IF ZERO
62 INSERT SIGN
62 INSERT CHARACTER SUPPRESS
63 INSERT CHARACTER ONCE
63 INSERT CHARACTER J TIMES
63 INSERT CHARACTERS IMMEDIATE
63 MOVE ALPHABETICS
63 MOVE NUMERICS
63 MOVE CHARACTERS
63 MOVE NUMERIC WITH ZERO SUPPRESSION
64 MOVE DIGIT WITH OVERPUNCH
64 MOVE FLOAT
64 END FLOAT
64 END EDIT

CHAPTER IV INPUT/OUTPUT

1 INTRODUCTION

1 OPERATION OF I/O DEVICES

2 PRIORITY INTERRUPTS

3 DATA CHANNEL

3 CODING AIDS

3 I/O INSTRUCTIONS
3 DATA IN A
3 DATA IN B
4 DATA IN C
4 DATA OUT A
4 DATA OUT B
4 DATA OUT C
4 I/O SKIP
4 NO I/O TRANSFER

5 CENTRAL PROCESSOR FUNCTIONS
5 INTERRUPT ENABLE
5 INTERRUPT DISABLE
5 INTERRUPT ACKNOWLEDGE
5 MASK OUT
7 VECTOR ON INTERRUPTING DEVICE CODE
8 READ SWITCHES
8 I/O RESET
8 HALT
8 CPU SKIP

9 ERROR CHECKING AND CORRECTION
9 Method of Operation
9 ENABLE ERCC
10 READ MEMORY FAULT ADDRESS
10 READ MEMORY FAULT CODE
10 ERCC Feature Memory Fault Codes

11 REAL TIME CLOCK
11 SELECT RTC FREQUENCY

11 POWER FAIL/AUTO-RESTART
11 SKIP IF POWER FAIL FLAG IS ONE
11 SKIP IF POWER FAIL FLAG IS ZERO

CONTENTS CONTINUED

CHAPTER V FRONT PANEL

1 INTRODUCTION

1 CONSOLE SWITCHES
1 Reset-Stop
2 Deposit-Examine
2 Exam-Exam Nxt
2 Inst-u/Inst
2 PR Load-Exec
2 Start-Cont
2 Dep-Dep Next
3 Address Compare
3 Power

4 PROGRAM LOADING
5 BOOTSTRAP LOADER

APPENDICES APPENDICES

APPENDIX A
I/O DEVICE CODES

APPENDIX B
OCTAL AND HEXADECIAML CONVERSION

APPENDIX C
ASCII CHARACTER CODES

APPENDIX D
DOUBLE PRECISION ARITHMETIC

APPENDIX E
COMPATIBILITY WITH NOVA LINE COMPUTERS

APPENDIX F
INSTRUCTION EXECUTION TIMES

APPENDIX G
USE OF THE VECTOR INSTRUCTION

APPENDIX H
INSTRUCTION USE EXAMPLES

APPENDIX I
S/200 And C/300 MAP

CHAPTER I
THE ECLIPSE LINE OF COMPUTERS

INTRODUCTION

The Data General Corporation ECLIPSE line of computers are general purpose, eight accumulator, stored-program computers with a word length of 16 bits. The maximum addressable amount of main memory for an ECLIPSE computer without the MAP feature is 65,536 8-bit bytes. If the MAP feature is installed, the maximum addressable amount of main memory is 524,288 bytes. Four of the accumulators are 16 bits in length and are used for arithmetic and logical operations. Two of these accumulators can also be used as index registers. The remaining four accumulators are 64 bits in length and are used for floating point arithmetic operations. Memory can be addressed either directly or by using indirect addresses. Chains of indirect address can be of any length. A direct memory access (DMA) data channel is provided to enable rapid data transfer between main memory and peripheral devices.

The ECLIPSE line of computers is made up of the S series and the C series. The S series consists of the ECLIPSE S/100, S/200, and S/230 computers. The C series consists of the ECLIPSE C/300 and C/330 computers. While these computers differ in specifics such as available features, they all share the same general architecture. This means that, in general, hardware is compatible across the entire line. To a somewhat lesser degree, software is also compatible across the line.

FEATURES OF THE ECLIPSE LINE OF COMPUTERS

The extensive capabilities of the ECLIPSE line of computers are a result of the features which have been designed as integral parts of the computer. These features allow the ECLIPSE line of computers to be used effectively in all types of system applications such as instrumentation and control, communications, computation, and data processing. The features of the ECLIPSE line of computers are summarized below.

Powerful Basic Instruction Set

The basic instruction set for the ECLIPSE line of computers contains instructions that perform fixed point arithmetic between accumulators, including multiply and divide; transfer of operands between accumulators and main storage; logical operations between accumulators; logical operations on bits and bytes both in memory and between accumulators; and data movement between memory locations.

Stack

A Last-In/First-Out (LIFO) or push-down stack is maintained by the processor. This feature provides a convenient method for the saving of return information and passing arguments between subroutines. The stack also provides an expandable area for the temporary storage of variables and intermediate results. A fast and efficient method of changing stacks is also provided so that a priority interrupt handler can make maximum use of the stack feature.

Floating Point

The floating point feature of the ECLIPSE line of computers allows the manipulation of both single precision (32 bits) and double precision (64 bits) floating point numbers. Single precision gives 6-7 significant decimal digits, while double precision gives 15-17 significant decimal digits. The decimal range of a floating point number is approximately 5.4×10^{-79} to $7.2 \times 10^{+75}$ in either precision.

Four separate 64-bit floating point accumulators are available to do floating point arithmetic. While the first operand of a floating point arithmetic instruction is always in one of the floating point accumulators, the second operand can either be in a floating point accumulator or fetched from memory. In addition to the standard arithmetic functions, instructions are available that compare two floating point numbers and set a condition code, or that test a floating point number for positivity

or negativity and conditionally skip upon the result of the test. The four floating point accumulators and the associated status bits can be pushed onto or popped off of the stack by one instruction. The floating point feature has been designed using the latest advances in technology for floating point computation. This makes the operation of the floating point feature extremely fast. In addition, the floating point feature operates in parallel with the rest of the central processor so that floating point computations can be performed simultaneously with fixed point computations.

The floating point feature is available on the ECLIPSE S/200 computer.

Commercial Instruction Set

The commercial instruction set feature of the ECLIPSE line of computers allows the processor to perform operations on data types commonly found in the commercial environment. Instructions are included that can move strings of bytes from one portion of memory to another; that can compare one string of bytes to another string of bytes and return an indicator which reflects whether one string is greater than, less than, or equal to the other; and that can translate a string of bytes from one representation to another depending upon a table of translation values. There is an instruction that can scan a string of bytes looking for a delimiter or one of a number of delimiters.

In addition to the string instructions, there are instructions in the commercial set that deal with decimal numbers in both packed and unpacked forms. These instructions operate with the Extended Arithmetic Processor (EAP) and allow the programmer to use floating point instructions to manipulate decimal numbers without losing any accuracy to round-off error. The Extended Arithmetic Processor possesses all the instructions and accumulators associated with the floating point feature plus the ability to convert numbers from their decimal representation to floating point representation and from floating point back to decimal. Instructions are included that can load and store decimal numbers having from 1 to 32 digits.

Finally, the commercial instruction set contains a powerful editing instruction that can convert a decimal number in either packed or unpacked form to a string of bytes under the control of an edit sub-program. This edit sub-program can perform many different operations on the number and its destination field including leading zero suppression, leading or trailing signs, floating fill characters, punctuation control, and insertion of text into the destination field.

The commercial instruction set and the EAP are features of the C series of ECLIPSE computers and are not available on computers in the S series.

Memory Allocation and Protection

The memory allocation and protection (MAP) feature of the ECLIPSE line of computers performs logical-to-physical address translation. Physical memory is allocated to a user in blocks of 2048 bytes and up to 32 such blocks can be allocated to a user at any one time. The same block of physical memory can be allocated to more than one user. This allows the sharing of procedure or data areas. The blocks of memory allocated to a user do not have to be contiguous.

The address translation function which correlates a logical address to the corresponding allocated physical memory address is called an "address map". The MAP feature is capable of holding three address maps at a single time. Two of the address maps are user address translation functions. The third address map translates addresses for the data channel. Only one user address map can be active at a time, but the data channel address map can be active at any time.

In addition to translating addresses, the MAP feature also performs various protection functions. A user is allowed to access only those blocks of memory allocated to him. This ensures that a user does not reach out of his own areas of memory for either instructions or data. Blocks of memory allocated to a user may be write-protected so that the user may not modify them. This allows blocks of memory containing constants or non-self-modifying procedures to be shared between users.

Input/Output devices can be declared accessible or inaccessible to a user on an individual device code basis. This allows any device to be controlled by the operating system or dedicated to a user, depending upon user requirements. Chains of indirect addresses that go deeper than sixteen levels can be detected and inhibited. This protects the system from becoming disabled by an indirection loop. Each of these protection functions can be enabled separately so the operating system can handle users with widely differing requirements. The MAP feature also allows the implementation of the LOAD EFFECTIVE ADDRESS instruction. This instruction allows the user to load the logical address of any memory location into an accumulator. This reduces the amount of memory that must be set aside to hold addresses and greatly reduces the number of instructions required to perform address arithmetic.

The MAP feature is available on the ECLIPSE S/200, S/230, C/300, and C/330 computers.

Extended Operation

The extended operation (XOP) feature of the ECLIPSE line of computers provides the user with a fast and general method of transferring control to called procedures. By issuing one instruction,

all relevant return information is placed on the stack and the address of the called procedure is retrieved from a user-constructed table of procedure addresses. After the address has been retrieved from the table, control is transferred to the procedure. There are two EXTENDED OPERATION instructions. Together, they are capable of transferring program control to one of 48 separate procedures.

Writeable Control Store

The writeable control store (WCS) feature of the ECLIPSE line of computers operates with the XOP feature to allow the user to implement his own specialized instructions. WCS is 256 56-bit words of extremely fast semiconductor memory. The 56-bit words contain instructions for controlling the elementary data paths of the computer. Instructions are placed in WCS by the user with the aid of input/output instructions. One of the two EXTENDED OPERATION instructions is used for transferring control to WCS routines. Because an ENTER WCS instruction can transfer control to one of 16 procedures, up to 16 instructions can be implemented at a time. WCS is a sophisticated feature and a full treatment is beyond the scope of this manual. WCS is completely described in "Microprogramming With the ECLIPSE Computer WCS Feature" (DGC 014-000045).

Writeable Control Store is a feature of the S series of ECLIPSE computers and is not available on computers in the C series.

Error Checking and Correction

The error checking and correction (ERCC) feature of the ECLIPSE line of computers provides the capability to detect and correct any single bit error in a word read from main memory. In an ERCC memory, a 5-bit check field is appended to each 2-byte word. The contents of this check field are constructed by a hardware encoder from the sixteen bits of the corresponding word. The check field is written each time the word is written and is checked each time the word is read. The code for the check field is such that all single-bit errors in either the data portion or check field portion of a memory location are detected. When an error is detected, the incorrect bit is corrected and the entire location in memory is rewritten before the data is passed along from the memory to the central processing unit. If desired, the ERCC feature can interrupt the central processor upon finding a memory error. This allows a record to be kept of memory errors.

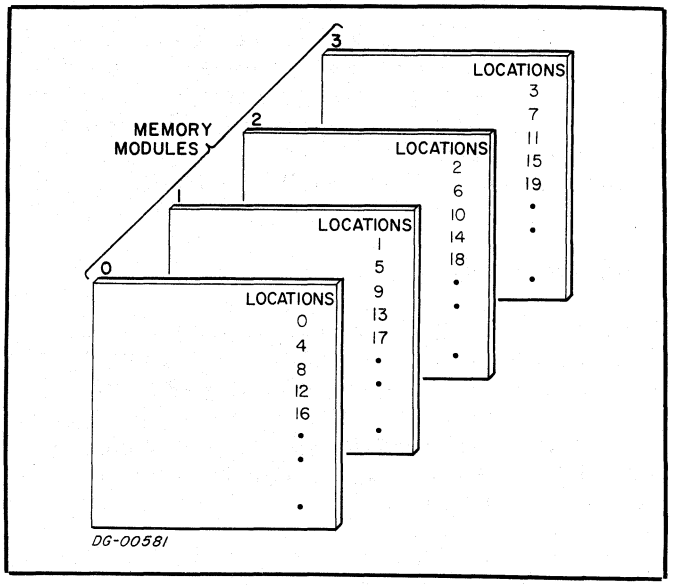
Memory Features

In addition to the ERCC feature, the ECLIPSE line of computers has other memory features.

In order to increase memory availability and reduce memory module contention, core memories may be interleaved. Interleaving is the process whereby consecutive memory locations are placed in different memory modules. This means that if consecutive memory locations are being referenced, a different memory module is referenced for each location. In this way, memory fetches can be overlapped. In a two-way interleaved system, the odd addresses are in one module and the even addresses are in the other module. In a four-way interleaved system, such as the one shown in the illustration, four consecutive locations reside in four different modules. Two-way, four-way, and eight-way interleaving are available. Different levels of interleaving may be mixed in the same system.

To increase memory speed, modules of semiconductor memory are available. Semiconductor memories may be interleaved in either two- or four-way schemes with other semiconductor memories.

All memories for the ECLIPSE line of computers are asynchronous. This allows the central processor to function at full speed and wait for the memory to respond only when absolutely necessary.



Power Fail/Auto-restart

The power fail/auto-restart feature of the ECLIPSE line of computers 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 portion of the feature senses the imminent loss of power and interrupts the processor. The interrupt service routine can then use this delay to store the contents of the accumulators, the program restart address, and other information that will be needed to restart the system. One to two milliseconds is enough time to execute 1,000 to 1,500 instructions on the ECLIPSE list of computers so there is more than enough time to perform the power fail routine.

When power is restored, the action taken by the auto-restart portion of the feature depends upon the position of the power switch on the front panel. If the switch is in the "on" position, the processor remains stopped after power is restored.

If the switch is in the "lock" position, then 222 milliseconds after power is restored, the processor executes the instruction contained in the first location of main memory, restarting the interrupted system.

Real-time Clock

The real-time clock feature of the ECLIPSE computer generates a sequence of pulses that is independent of the timing of the processor. The clock will interrupt the system at one of four program-selectable frequencies. The frequencies are: ac line frequency, 10Hz, 100Hz, and 1000Hz.

Input/Output Bus

The input/output (I/O) bus is that portion of the ECLIPSE line of computers system that carries commands and data between the computer and various peripheral devices connected to it. The bus is made up of a six-line device selection network, interrupt circuitry, command circuitry, and sixteen data lines.

Device Addressability

Each I/O device in an ECLIPSE line of computers system is connected to the six-line device selection network in such a way that each device will only respond to commands that contain its own device code. The fact that the selection network is made up of six lines gives $2^6 = 64$ unique device codes. Five of these codes are reserved for specific ECLIPSE line of computers features and functions, but there are still 59 device codes available for use with I/O devices.

Interrupt Capability

The interrupt circuitry contained in the I/O bus provides the capability for any I/O device to interrupt the system when that device requires service. When a device requests an interrupt, the processor automatically transfers program control to the main interrupt service routine. This routine can either poll all the I/O devices in the system to find out which one initiated the interrupt or the routine can use one of two instructions to identify the source of the interrupt.

The INTERRUPT ACKNOWLEDGE instruction returns the device code of the interrupting device. The VECTOR ON INTERRUPTING DEVICE CODE instruction not only returns the device code of the interrupting device, but also saves return information on the stack and transfers program control to the correct service routine for the device.

The interrupt circuitry of the ECLIPSE line of computers also contains the capability to implement up to sixteen levels of priority interrupts. This is done with a 16-bit priority mask. Each level of device priority is associated with a bit in this mask. In order to suppress interrupts from any priority level, the corresponding bit in the mask is set to 1. In addition to saving return information and transferring control, the VECTOR ON INTERRUPTING DEVICE CODE instruction updates this mask, and therefore makes the implementation of a priority interrupt system a straightforward procedure.

Data Channel

Handling data transfers between external devices and memory under program control requires an interrupt plus the execution of several instructions for each word transferred. To allow greater transfer rates, the I/O bus contains circuitry for a direct memory access (DMA) data channel through which a device, at its own request, can gain direct access to memory using a minimum of processor time. At the maximum input rate of 1,250,000 words per second or at the maximum output rate of approximately 715,000 words per second, the data channel effectively stops the processor, but at lower rates processing continues while data is being transferred.

Ease of Interfacing

Due to the straightforward logic and general design of the I/O bus on the ECLIPSE line of computers, customer provided or customer designed I/O devices may be interfaced easily to an ECLIPSE line of computers system.

Input/Output Devices

A comprehensive array of I/O devices is available from Data General for the ECLIPSE line of computers. This wide choice of devices, ranging from teletypewriters to line printers to video display for man-machine interaction; and from paper tape to magnetic tape to fixed and moving-head discs for data storage allows a wide spectrum of possible configurations. Also available are various multiplexors and telecommunications adapters, including an IBM 360/370 interface.

Software

The ECLIPSE line of computers is fully supported by proven Data General software. Because the ECLIPSE line of computers is compatible with the NOVA line of computers, the programming systems available in the past have been easily altered to take advantage of the processing advancements provided by the expanded instruction set of the ECLIPSE line of computers. These alterations have been accomplished without sacrificing any of the desirable features of these systems.

Languages

In addition to an assembler and a macroassembler, there are powerful higher-level language processors available for use with the ECLIPSE line of computers. Language processors such as ALGOL, EXTENDED BASIC, and FORTRAN 5 have been updated for the ECLIPSE line of computers to ease the job of implementing applications systems.

Operating Systems

Several operating systems are available for the ECLIPSE line of computers:

- Stand-alone Operating System (SOS)
  Real-Time Operating System (RTOS)
  Real-time Disc Operating System (RDOS)
  Advanced Operating System (AOS)

SOS and RDOS software are designed for the small to medium-size systems, while AOS software has been updated to take full advantage of all the features embodied in the ECLIPSE line of computers.

Conclusion

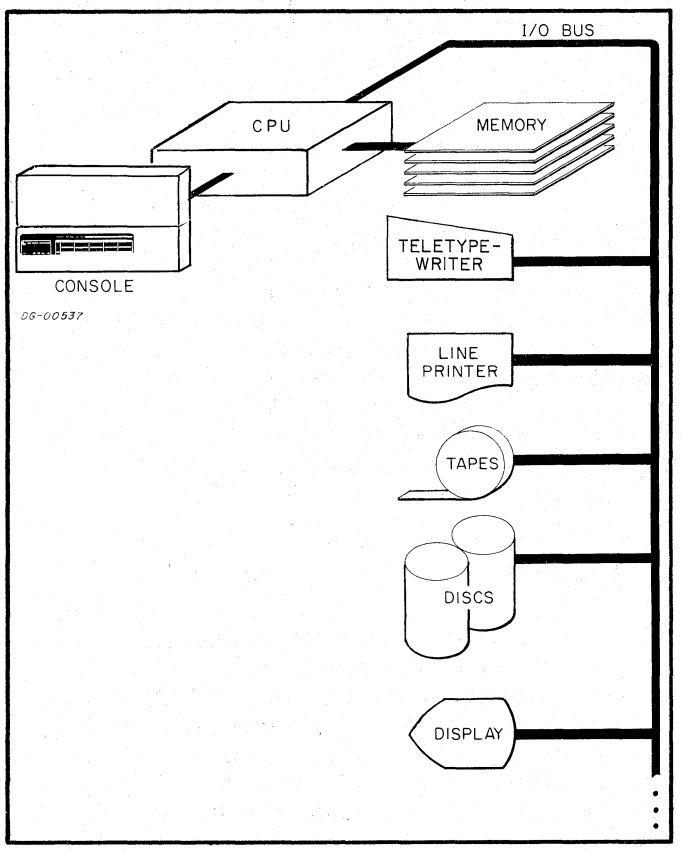
The comprehensiveness of the internal features, software and I/O devices available with the ECLIPSE line of computers ensures that ECLIPSE line of computers systems can be effectively configured to satisfy the unique and specific needs of instrumentation and control, communications, computation, and data processing applications.

CHAPTER II

INTERNAL STRUCTURE

INTRODUCTION

The basic structure of an ECLIPSE line data processing system consists of a central processing unit (CPU), some amount of main memory, the I/O bus, the I/O devices connected to the I/O bus, and a console which is on the front panel of the main computer chassis.



Due to the general-purpose design of the ECLIPSE computer, the type, size, and number of memory modules and I/O devices have no effect upon the internal logical structure of the CPU. The CPU is made up of the fixed-point arithmetic and logical unit, the floating point arithmetic unit, the MAP feature, the WCS feature, and the real-time clock feature. In addition, there are eight accumulators. Four of these are 16 bits in length and are used by the fixed point unit. The other four are 64 bits in length and are used by the floating point unit. This chapter deals with the addressing of information and the logical representation of information within the CPU, and is unaffected by those portions of the system outside the CPU.

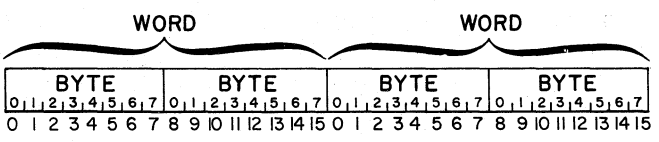
INFORMATION FORMATS

The basic piece of information within the processor is the binary digit, or "bit". A bit is capable of representing only two quantities, 0 and 1. However, a bit cannot represent both these values at the same time. At any one point in time, a bit can either represent a 0 or a 1, never both.

The normal unit of information within the CPU is the "byte". A byte is made up of 8 bits. Because each bit is capable of representing two quantities, a byte is capable of representing $2^8 = 256$ different quantities. Two bytes may be combined to produce a 16-bit unit called a "word". A word can represent $2^{16} = 65,536$ different quantities. I/O devices transfer information in units of bits, bytes, words, or multiples of words called "records", depending upon the device.

Bit Numbering

In order to avoid confusion when talking about the information contained in bytes and words, the bits that make up these units of information are numbered from left to right, with the leftmost (high-order) bit always numbered bit 0. The numbering extends to the right and is always carried out in the decimal number system. The rightmost (low-order) bit in a byte is bit 7. The rightmost bit in a word is bit 15.



Octal Representation

Because talking about the binary data contained in bytes and words would quickly become awkward and confusing if each bit were described, the octal representation of binary information will be used in this manual. To convert a piece of binary information to its octal representation, the bits in the quantity are separated into groups of three bits each, starting from the right and proceeding to the left. If the number of bits to be represented is not evenly divisible into groups of three, the leftmost group will contain one or two bits. Each group of bits can now be represented by one of eight different symbols. The digits 0-7 are used to represent the quantities 0-7. Each encoded digit is called an octal digit. Because each group of bits can contain any one of 8 values, this representation is sometimes called "base 8" representation.

Another way to represent binary information is the hexadecimal or "hex" representation. In hexadecimal, the bits in the quantity are separated into groups of four bits each and each group can be represented by one of 16 different symbols. The digits 0-9 are used to represent the quantities 0-9. The letters A-F are used to represent the quantities 10-15. Because each group of bits can contain any one of 16 values, this representation is sometimes called "base 16" representation.

The following table gives the correspondence between the various representations.

DECIMAL	BINARY	HEX	BINARY	OCTAL
0	0000	0	000	0
1	0001	1	001	1
2	0010	2	010	2
3	0011	3	011	3
4	0100	4	100	4
5	0101	5	101	5
6	0110	6	110	6
7	0111	7	111	7
8	1000	8	1 000	10
9	1001	9	1 001	11
10	1010	A	1 010	12
11	1011	B	1 011	13
12	1100	C	1 100	14
13	1101	D	1 101	15
14	1110	E	1 110	16
15	1111	F	1 111	17

Our normal decimal numbering system is sometimes called "base 10" representation. Because it is sometimes possible to confuse numbers written in hex or octal with those written in decimal, a subscript denoting the base will be used in cases where confusion might occur. The following examples illustrate this convention.

$$64_{10} = 40_{16} = 100_8$$
$$87_{10} = 57_{16} = 127_8$$
$$63_{10} = 3F_{16} = 77_8$$

In the last example, it is obvious that 3F is a number written in hex, but the subscript is included to erase any possible doubts.

Conversion tables for hex to decimal and octal to decimal are contained in Appendix B of this manual.

Character Codes

Within the processor, all information is represented by binary quantities. The CPU does not recognize certain bit combinations as characters and certain other bit combinations as numbers. Sooner or later, however, this information must be transferred outside the computer in some form easily understood by humans. For this reason, some standard correspondence must be made between certain bit combinations and printable symbols. The code used to implement this correspondence in I/O devices available with the ECLIPSE line of computers is called the American Standard Code for Information Interchange (ASCII). This code can represent 95 printable symbols plus 33 control functions. A complete table of codes and their corresponding characters can be found in Appendix C of this manual.

Information Representation

Even though the CPU does not intrinsically recognize one information type from another, the different instructions in the instruction set expect that the information to be operated on will be in a specific format. In general, there are four different, basic information formats. They are integers, floating point numbers, logical quantities, and decimal numbers.

Integers

Integers can be represented as either signed or unsigned numbers and carried in either single or multiple precision. Single precision integers are two bytes long, while multiple precision integers are four or more bytes long. Unsigned integers use all the available bits to represent the magnitude of the number. A single two-byte word can represent any unsigned number in the inclusive range 0 to 65,535. Two words taken together as an unsigned, double precision integer can represent any number in the inclusive range 0 to 4,294,967,295.

For signed operations, the two's complement numbering system is used. In this system, the leftmost or high-order bit is used as a sign bit. If the sign bit is 0, the number is positive and the remainder of the bits in the number represent the magnitude of the number as described above. If the sign bit is 1, the number is negative and the remainder of the bits represent the two's complement of the magnitude of the number.

To create the negative of a number in the two's complement scheme, complement all the bits of the number including the sign bit. After the complementing process is finished, add 1 to the rightmost or low-order bit. If the two's complement of a negative number is formed, the result will be the corresponding positive number. There is only one representation for zero in two's complement arithmetic: it is the number with all bits zero. Forming the two's complement of zero will produce a carry out of the high-order bit and leave the number with all bits zero.

Examples:

To form the negative of 4:

4 =	0	000	000	000	000	100
complement =	1	111	111	111	111	011
add 1 +						1
-4 =	1	111	111	111	111	100

To form the negative of 1715_8:

1715_8 =	0	000	001	111	001	101
complement =	1	111	110	000	110	010
add 1 +						1
-1715_8 =	1	111	110	000	110	011

To form the negative of -1715_8:

-1715_8 =	1	111	110	000	110	011
complement =	0	000	001	111	001	100
add 1 +						1
1715_8 =	0	000	001	111	001	101

To form the negative of 0:

0 =	0	000	000	000	000	000
complement =	1	111	111	111	111	111
add 1 +						1
0 =	0	000	000	000	000	000

Note that 0 is a positive number, i.e., its sign bit is 0.

Because the two's complement scheme has only one representation for 0, there is always one more negative number than there are non-negative numbers. The most negative number is a number with a 1 in the sign bit and all other bits 0. The positive value of this number can not be represented in the same number of bits as used to represent the negative number.

A single two-byte word can represent any signed number in the inclusive range -32,768 to +32,767. Two words taken together as a signed, double precision integer can represent any number in the inclusive range -2,147,483,648 to +2,147,483,647.

It is a property of numbers using the two's complement scheme that addition and subtraction of signed numbers are identical to addition and subtraction of unsigned numbers. The CPU just treats the sign bit as the most significant magnitude bit. This does not work for multiplication and division, however, so the ECLIPSE line instruction set contains both signed and unsigned multiply and divide instructions.

Floating Point

The floating point feature of the ECLIPSE line of computers allows operations on signed numbers having a much larger range than those normally represented as integers. It would take a 16-word multiple precision integer to represent the range of an ECLIPSE line floating point number. Since floating point numbers occupy either two words for single precision or four words for double precision, and the floating point feature is much faster than multiple precision integer software routines, floating point arithmetic is used when numbers having a large range must be manipulated.

A floating point number is made up of three parts: the sign, the exponent, and the mantissa. The value of a floating point number is defined to be:

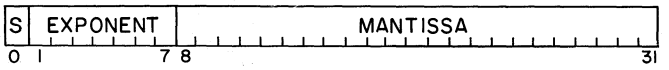
(MANTISSA) X (16 RAISED TO THE TRUE VALUE OF THE EXPONENT FIELD)

The number is signed according to the value of the sign bit. If the sign bit is 0, the number is positive; if the sign bit is 1, the number is negative.

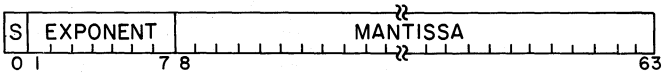
Floating point numbers are represented internally by either 32 bits (single precision) or 64 bits (double precision).

The formats are shown below:

Single Precision



Double Precision



Bit zero is the sign bit: 0 for positive, 1 for negative.

Bits 1-7 contain the exponent. This is the power to which 16 must be raised in order to give the correct value to the number.

We use "excess 64" representation in the exponent field to obtain both positive and negative exponents. This means that the value in the exponent field is 64 greater than the true value of the exponent. If the exponent field is zero, the true value of the exponent is -64. If the exponent field is 64, the true value of the exponent is 0. If the exponent field is 127, the true value of the exponent is 63.

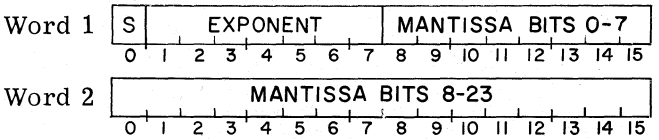
Bits 8-31 for single precision and bits 8-63 for double precision contain the mantissa. This means that bit 8 of the floating point number is bit 0 of the mantissa. The mantissa is always a positive fraction greater than or equal to 1/16 and less than 1. The "binary point" can be thought of as being just to the left of bit 8. Continuing this concept then, bit 8 represents the value 1/2, bit 9 represents the value 1/4, bit 10 represents the value 1/8, and so on.

In order to keep the mantissa in the range of 1/16 to 1, the results of floating point arithmetic are "normalized". Normalization is the process whereby the mantissa is shifted left one hex digit at a time until the high-order four bits represent a nonzero quantity. For every hex digit shifted, the exponent is decreased by one. Since the mantissa is shifted four bits at a time, it is possible for the high-order three bits of a normalized mantissa to be zero.

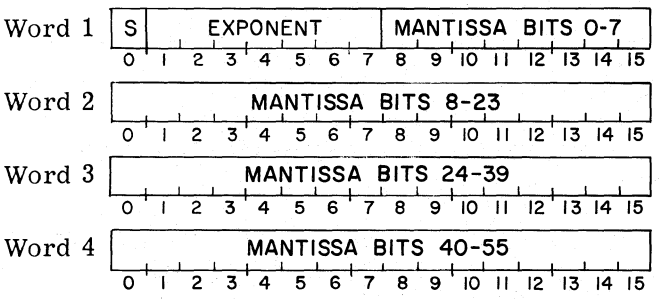
Zero is represented by a floating point number with all bits zero. This is true for both single and double precision. This is known as "true zero". When a calculation results in a zero mantissa, the floating point processor automatically converts the number to a true zero. Note that true zero is positive. It is not possible to obtain negative zero as the result of a calculation.

Floating point operands in memory are represented by two words for single precision and by four words for double precision. The formats are shown below:

Single Precision



Double Precision



Logical Quantities

Logical operations in the ECLIPSE line can be performed upon individual bits, bytes, or words. When using the logical operations, quantities operated on are treated as unstructured binary quantities. The number of bits, bytes, or words operated upon depends on the particular instruction.

Decimal Numbers

Decimal numbers may be represented internally in two ways, unpacked decimal and packed decimal. In unpacked decimal, the number is made up of a string of ASCII characters and the sign, if present, may appear in one of four places. The sign of the number may be indicated by a leading or trailing byte which contains the ASCII code for plus ($2B_{16}$) or minus ($2D_{16}$). Alternatively, either the high-order digit or the low-order digit of the number may indicate the sign in addition to carrying a digit of the number. The table below gives the correspondence between certain ASCII characters and the sign and digit values that they carry.

SIGN VALUE	DIGIT VALUE	ASCII CHARACTER	HEX CODE
+	0	ø + 0 {	20, 2B, 30, 7B
+	1	1 A	31, 41
+	2	2 B	32, 42
+	3	3 C	33, 43
+	4	4 D	34, 44
+	5	5 E	35, 45
+	6	6 F	36, 46
+	7	7 G	37, 47
+	8	8 H	38, 48
+	9	9 I	39, 49
-	0	- }	2D, 7D
-	1	J	4A
-	2	K	4B
-	3	L	4C
-	4	M	4D
-	5	N	4E
-	6	O	4F
-	7	P	50
-	8	Q	51
-	9	R	52

DG-01288

The digits that are not carrying the sign must be valid ASCII characters for the digits 0-9 (30_{16}-39_{16}) or spaces (20_{16}). A space has the same value as a zero.

Examples:

In the following examples, the hex value of a byte is shown inside the box; the corresponding ASCII character is shown beneath the box.

	Byte	Byte	Byte	Byte	Byte
+2,048 (leading sign)	2B	32	30	34	38
	+	2	0	4	8
-1,756 (trailing sign)	31	37	35	36	2D
	1	7	5	6	-
+1,850 (high-order sign)	41	38	35	60	
	A	8	5	0	
-3,970 (low-order sign)	33	39	37	7D	
	3	9	7	}	

For packed decimal, each digit of the decimal number occupies one hex digit. The sign is specified by a trailing hex digit. The number must start and end on a byte boundary. In other words, the number cannot start or end halfway through a byte. This means that a packed decimal number will always consist of an odd number of digits followed by the sign. The sign must be either C_{16} for plus or D_{16} for minus. The only valid codes for digits are $0\text{-}9_{16}$.

Examples:

In the following examples, the hex value of a digit is shown within the box; the corresponding decimal digit is shown beneath the box.

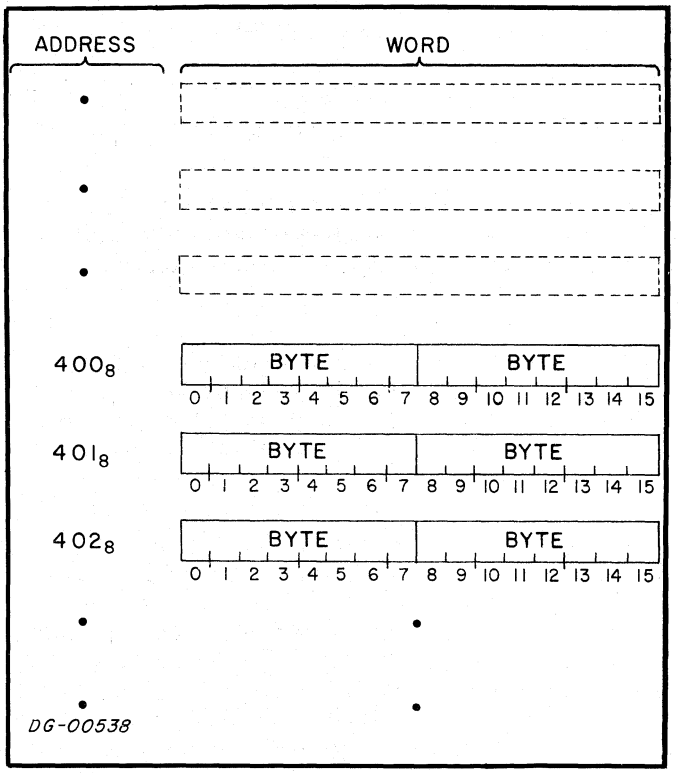
	Byte		Byte		Byte	
+2,048	0	2	0	4	8	C
	0	2	0	4	8	+
+32,456	3	2	4	5	6	C
	3	2	4	5	6	+
-1,756	0	1	7	5	6	D
	0	1	7	5	6	-
-25,989	2	5	9	8	9	D
	2	5	9	8	9	-

INFORMATION ADDRESSING

The information formats described in the preceding section give a way of representing different types of data within the CPU. Operations cannot be performed upon these data types, however, unless they can be addressed by the CPU. The address of a piece of information is its location in main memory. Once the CPU knows the address of a piece of information, the desired operation can be performed.

Word Addressing

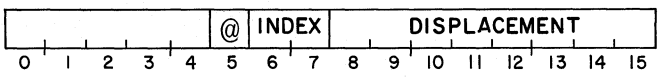
Main memory is partitioned into 2-byte words, and each word has an address. The first word in memory has the address 0. The next word has the address 1, the next word has the address 2, and so on. Word addressing is used to address integers, floating point numbers, and logical quantities that are formatted in units of words.



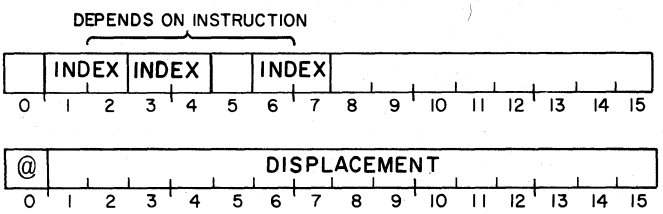
Effective Address Calculation

The instructions in the ECLIPSE line instruction set that directly reference memory using word addressing fall into two classes. The "short" class of instructions uses 11 bits in the instruction word to define the address. The "extended" class of instructions uses two bits in the instruction word plus the 16 bits of the word following the instruction to define the address. These bits do not directly specify the address, but are used in a calculation which results in the address of the desired word. The resultant address is called the "effective address" or "E", and the calculation is called the "effective address calculation".

For the short class, 11 bits in the instruction are used to define the effective address. Bit 5 is called the "indirect bit", bits 6 and 7 are called the "index bits" and bits 8-15 are called the "displacement bits".



For the extended class, 2 bits in the instruction plus the next word are used to define the effective address. Depending on the instruction, either bits 1 and 2 or bits 3 and 4 or bits 6 and 7 of the instruction are the index bits. In the next word, bit 0 is the indirect bit and bits 1-15 are the displacement bits.



If the index bits are 00, the displacement bits are treated as an unsigned number which is the address of a word in memory. This is called "absolute addressing".

If the index bits are 01, the displacement bits are treated as a signed, two's complement number which is added to the address of the word containing the displacement bits. This is called "relative addressing".

If the index bits are 10, accumulator 2 is used as an index register. If the index bits are 11, accumulator 3 is used as an index register. In this form of word addressing, known as "index register addressing", the displacement is treated as a signed, two's complement number which is added to the contents of the selected index register to produce a memory address. In index register addressing, the addition of the displacement to the contents of the index register does not change the value contained in the index register.

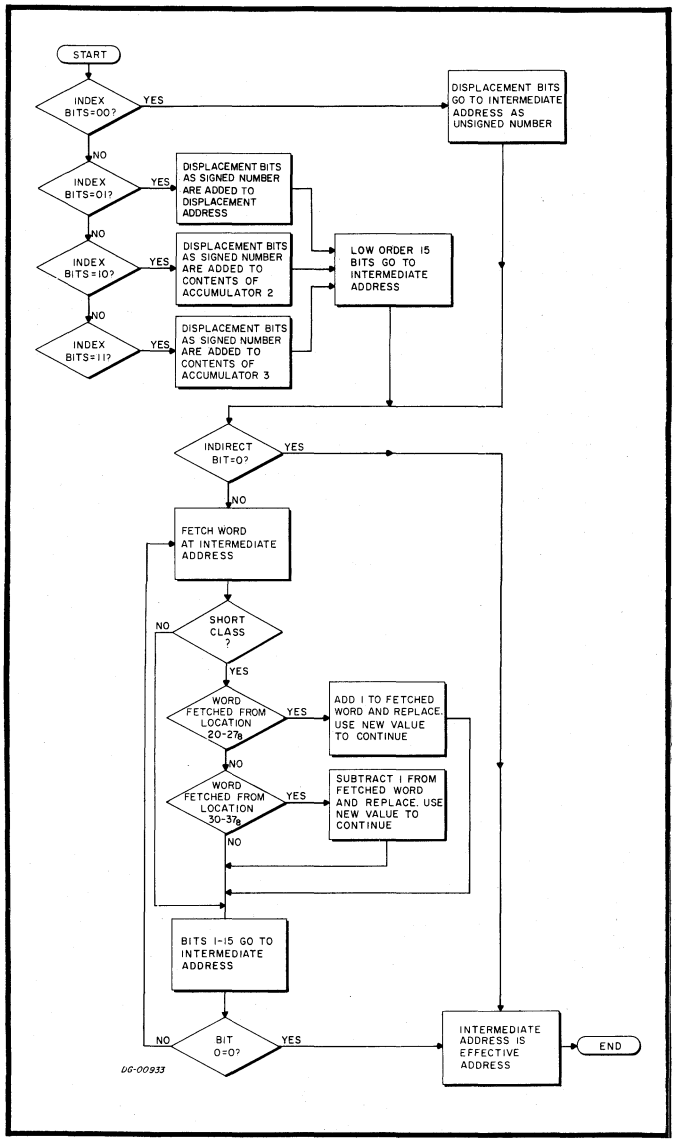
The result of the addition performed in relative addressing and index register addressing is "clipped" to 15 bits. In other words, the high-order bit of the result is set to 0. For example, if accumulator 2 is to be used as an index register and contains the number 077774_8, and the displacement bits contain the number 012_8, then the result of the addition would be 000006_8, not 100006_8.

After one of the three types of addresses has been computed from the index and displacement bits, the indirect bit is tested. If this bit is zero, the address already computed is taken as the effective address. If the indirect bit is one, the word addressed by the result of the index and displacement bits is assumed to contain an address. In this word bit 0 is the indirect bit and bits 1-15 contain an address. If bit 0 of the referenced word is 1, another level of indirection is indicated, and bits 1-15 contain the address of the next word in the indirection chain. The processor will continue to follow this chain of indirect addresses until a word is retrieved with bit 0 set to 0. Bits 1-15 of this word are taken to be the effective address.

For the short class of instructions, if an indirect address points to a location in the range $20\text{-}27_8$ (auto-increment locations), that word is fetched, the contents of the word are incremented by one and written back into the location. This updated value is then used to continue the addressing chain. If an indirect address points to a location in the range $30\text{-}37_8$ (auto-decrement locations), that word is fetched, the contents of the word are decremented by one and written back into the location. The updated value is then used to continue the addressing chain.

NOTE When referencing auto-increment and auto-decrement locations, the state of bit 0 before the increment or decrement is the condition upon which the continuation of the indirection chain is based. For example: If an auto-increment location contains 177777_8, and the location is referenced as part of an indirection chain, location 0 will be the next address in the chain. That is, the effective address will not be 0.

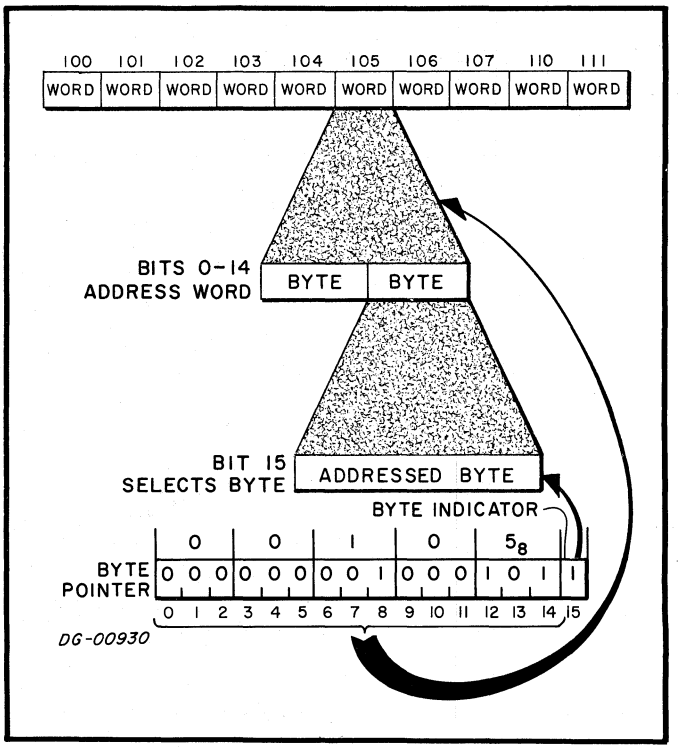
The auto-increment and auto-decrement feature only works with the following short class of instructions: LOAD, STORE, JUMP, JUMP TO SUBROUTINE, INCREMENT AND SKIP IF ZERO, DECREMENT AND SKIP IF ZERO, LOAD EFFECTIVE ADDRESS, and during the program interrupt cycle.



An effective address is always 15 bits in length. This means that an instruction which uses the effective address calculation can address any one of $32,768_{10}$ words. This gives rise to the concept of an "address space", which, in the ECLIPSE computer, contains 64K bytes or 32,768 2-byte words.

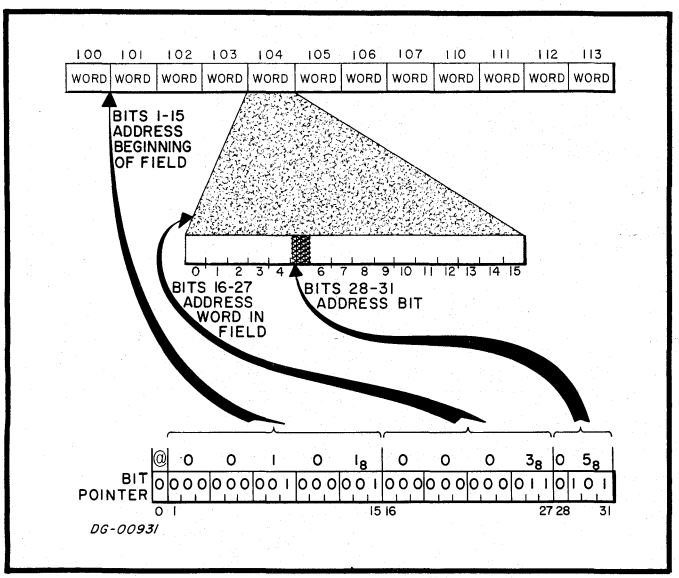
Byte Addressing

There are two instructions that directly reference bytes in memory. These instructions address bytes by using a "byte pointer". A byte pointer is a word in which bits 0-14 are the address in memory of a 2-byte word. Bit 15 of the byte pointer is the "byte indicator". If the byte indicator is 0, the referenced byte is the high-order (bits 0-7) byte of the word addressed by byte pointer bits 0-14. If the byte indicator is 1, the referenced byte is the low-order (bits 8-15) byte of the word addressed by byte pointer bits 0-14.



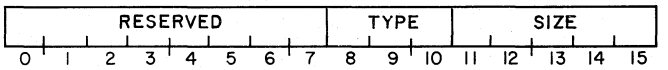
Bit Addressing

There are five instructions that directly reference bits in memory. These instructions address bits by using a 32-bit "bit pointer". Bit 0 of the bit pointer is the indirect bit. If this bit is 1, the indirection chain 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. Bits 1-15 contain the address of a word which is the beginning of a 4096 word field. Bits 16-27 of the bit pointer contain a 12-bit positive number, which, when added to the number contained in bits 1-15 of the bit pointer produce the address of the desired word within the field. None of this addition affects the original contents of the bit pointer. Bits 28-31 of the bit pointer contain an unsigned 4-bit number which is the number of the desired bit within the referenced word.



Commercial Instruction Set Addressing

The instructions in the commercial instruction set all use byte addressing to address their operands. Several of the instructions in this set also use an attribute specifier word to determine the data type of the operands. The format of the attribute specifier word is as follows:

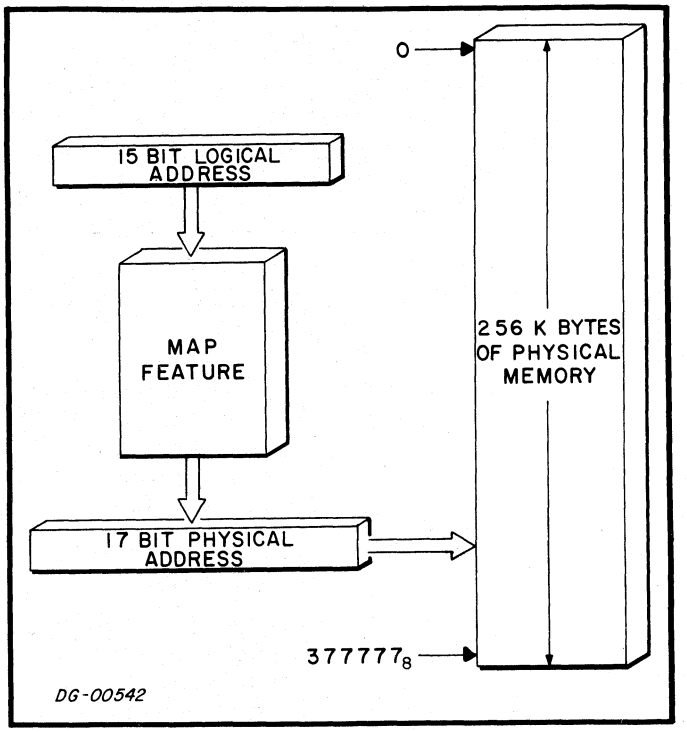


Bits	Name	Contents
0-7	---	Reserved for future use.
8-10	Type	Signify the type of the data as follows: 0 Unpacked decimal--low-order sign 1 Unpacked decimal--high-order sign 2 Unpacked decimal--trailing sign 3 Unpacked decimal--leading sign 4 Unpacked decimal--unsigned 5 Packed decimal 6 Two's complement integer--byte aligned 7 Floating point--byte aligned
		NOTE Data types 6 and 7 are the same as the two's complement integers and floating point numbers described under Information Representation except that they need not begin and end on a word boundary. Rather, they may begin and end half-way through a word.
11-15	Size	Signify the length of the data as follows: For all data types except type 5, this is the number of bytes in the number (including leading or trailing signs) minus 1. For data type 5, this is the number of digits in the number.

Addressing With The MAP Feature

The concept of an address space was introduced in the discussion of effective address calculation. The "program" or "logical" address space is that amount of memory that can be referenced by instructions in a program. The "physical" address space is the amount of physical memory that can be referenced by the CPU. If the MAP feature is not installed, the physical address space available to the ECLIPSE line of computers CPU is 64K bytes or 32K words, and the logical address space is equal to the physical address space. Obviously, if the system contains less than 64K of physical memory, the usable address space is reduced, but the maximum physical address space of the CPU without the MAP feature is 64K. With the MAP feature installed, the logical address space is still 64K, but the maximum physical address space is increased to 256K bytes.

Installation of the MAP feature has no effect on logical addressing. The addressing calculations remain the same. The MAP feature comes into play when the CPU tries to use a 15-bit address to reference memory. The MAP feature intercepts the memory reference and the 15-bit address. The MAP feature then translates the 15-bit address into a 17-bit address with the aid of address translation hardware and the logical-to-physical address translation functions that have been set up by the supervisor program. The resultant 17-bit address is used to reference memory.



RESERVED STORAGE LOCATIONS

In addition to the four accumulators, called AC0, AC1, AC2, and AC3, which have already been mentioned, there are thirty-two reserved storage locations in the ECLIPSE line of computers. These are locations in main memory that have special meaning for the CPU. The address of these locations, their names, and their functions are given in the table below. The notation "indirectable" means that bit 0 may be set to indicate that this is an indirect address.

LOCATION ADDRESS	LOCATION NAME	LOCATION FUNCTION
Octal		
0	I/O RETURN ADDRESS	Return address from I/O interrupt. Also 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.
4	VECTOR STACK POINTER	Address of the top of the VECTOR stack. Non-indirectable.
5	CURRENT MASK	Current interrupt priority mask.
6	VECTOR STACK LIMIT	Address of the last normally usable location in the VECTOR stack.
7	VECTOR STACK FAULT ADDRESS	Address of the VECTOR stack fault handler. Indirectable.
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. Non-indirectable.
41	FRAME POINTER	Address of the start of the current stack frame minus 1. Non-indirectable.
42	STACK LIMIT	Address of the last normally usable location in the stack.
43	STACK FAULT ADDRESS	Address of the stack fault handler. Indirectable.
44	XOP ORIGIN ADDRESS	Address of the beginning of the XOP table. Non-indirectable.
45	FLOATING POINT FAULT ADDRESS	Address of the floating point fault handler. Indirectable.
46	COMMERCIAL FAULT ADDRESS	Address of the commercial fault handler.
47		Reserved for future use.

PROGRAM EXECUTION

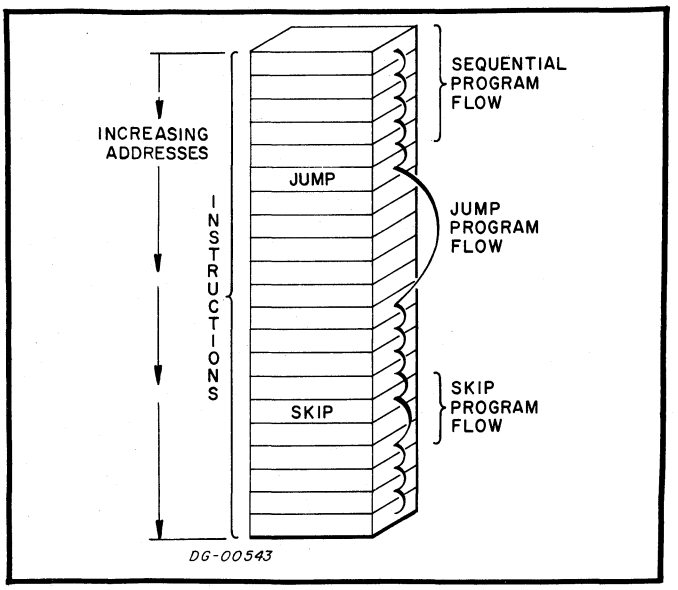
Programs for the ECLIPSE line of computers consist of sequences of instructions that reside in main memory. The order in which these instructions are executed depends on a 15-bit counter called the "program counter". The program counter always contains the address of the instruction currently being executed. After the completion of each instruction the program counter is incremented by one and the next instruction is fetched from this address. This method of operation is called "sequential operation" and the instruction fetched from the location addressed by the incremented program counter is called the "next sequential instruction".

Program Flow Alteration

Sequential operation can be explicitly altered by the programmer in two ways. Jump instructions alter program flow by inserting a new value into the program counter. Conditional skip instructions can alter program flow by incrementing the program counter an extra time if a specified test condition is true. In the case of a conditional skip instruction when the test condition is true, the next sequential instruction is not executed because it is not addressed. After either a jump instruction or a successful conditional skip instruction, sequential operation continues with the instruction addressed by the updated value of the program counter.

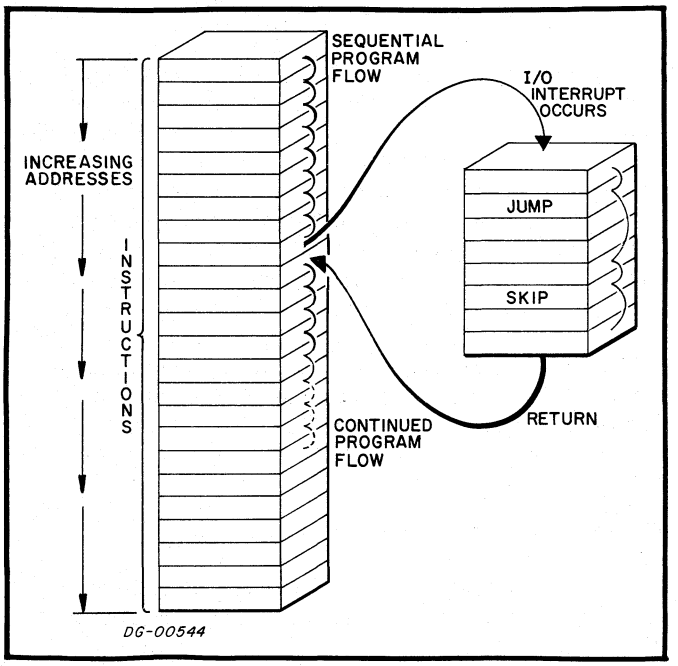
Note that there are some instructions in the instruction set that are 32 bits in length. It is not possible to skip over these instructions with a conditional skip instruction. If the programmer attempts to skip over one of these instructions with a conditional skip instruction, the second word of the 32-bit instruction will be executed as an instruction.

Because the program counter is 15 bits in length, it can address 32,768 separate memory locations. The next memory location after 77777_8 is location 0, and the location before 0 is location 77777_8. If the program counter rolls from 77777_8 to 0 in the course of sequential operation, no indication is given and processing continues with the location addressed by the updated value of the program counter.



Program Flow Interruption

The normal flow of a program may be interrupted by external or exceptional conditions such as I/O interrupts or various kinds of faults. In this case, the address of the next sequential instruction in the interrupted program is saved by the CPU so that the I/O handler or the various fault handlers can return control to the program at the correct point. Once the address of the next sequential instruction in the program has been placed in the program counter by the fault handler, sequential operation of the program resumes.



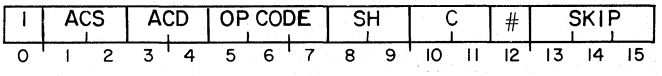
CHAPTER III
INSTRUCTION SETS

INTRODUCTION

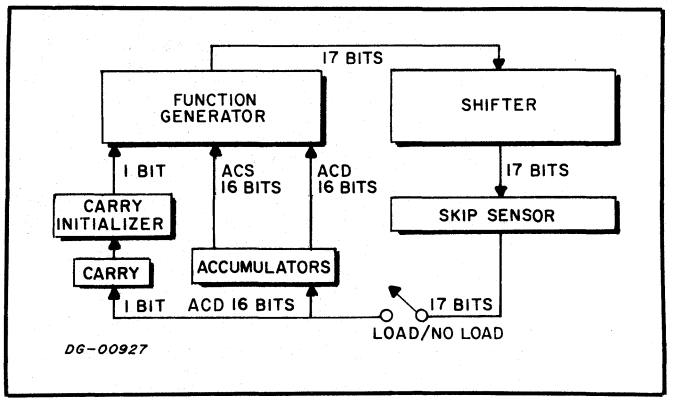
The instruction set implemented on the ECLIPSE line computer is divided into 10 instruction sets. There are instruction sets available for fixed point arithmetic, logical operations, byte manipulation, bit manipulation, data movement, stack manipulation, program flow alteration, floating point arithmetic, string and decimal number manipulation, and I/O operations. In addition, instruction sets which are a mixture of I/O instructions and machine instructions are available for programming the MAP feature, the XOP feature, the ERCC feature, the RTC feature, the power fail/auto-restart feature, and certain CPU functions.

INSTRUCTION FORMATS

The instruction set of the ECLIPSE line of computers is extremely bit-efficient. Therefore, the set does not break into convenient instruction formats. There are however, eight instructions which share a common format and utilize a specialized arithmetic unit. This format is called the "Two Accumulator-Multiple Operation" format.



In the Two Accumulator-Multiple Operation format instructions, bit 0 is 1, bits 1 and 2 specify the source accumulator, bits 3 and 4 specify the destination accumulator, bits 5-7 contain the operation code, bits 8 and 9 specify the action of the shifter, bits 10 and 11 specify the value to which the carry bit will be initialized, bit 12 specifies whether or not the result will be loaded into the destination accumulator, and bits 13-15 specify the skip test. Each instruction in this format utilizes an arithmetic unit whose logical organization is illustrated below.



Each instruction specifies two accumulators to supply operands to the function generator, which performs the function specified by bits 5-7 of the instruction. The function generator also produces a carry bit whose value depends upon three quantities: an initial value specified by the instruction, the function performed, and the result obtained. The initial value may be derived from the previous value of the carry bit, or the instruction may specify an independent value.

The 17-bit output of the function generator, made up of the carry bit and the 16-bit function result, then goes to the shifter. In the shifter, the 17-bit result can be rotated one place right or left, or the two 8-bit halves of the function result can be swapped without affecting the carry bit. The 17-bit output of the shifter can then be tested for a skip. The skip sensor can test whether the carry bit or the rest of the 17-bit result is or is not equal to zero. After the skip sensor has tested the shifter output it can be loaded into the carry bit and the destination accumulator. Note, however, that loading is not necessary. An instruction in this format can perform a complicated arithmetic and shifting operation and test the result for a skip without affecting the carry bit or either of the operands.

CODING AIDES

In the descriptions of the separate instructions, the general form of how the instruction is coded in assembly language is given along with the instruction format and the description of the instruction. The general form of how an instruction may be coded has the following format:

MNEMONIC<optional mnemonics> OPERAND STRING

The mnemonic must be coded exactly as shown in the instruction description. Some instructions have optional mnemonics that may be appended to the main mnemonic if the option is desired. The operand string is made up of the operands for the given instruction.

The symbols <> and ═ are used in this manual to aid in defining the instructions. These symbols are not coded; they act only to indicate how an assembly language instruction may be written. Their general definition is given below.

<> Indicates optional operands or mnemonics. The operand enclosed in the brackets (e.g., <#>) may be coded or not, depending on whether or not the associated option is desired.

═ Indicates specific substitution is required. Substitute the desired accumulator, address, name, number, or mnemonic.

The following abbreviations are used throughout this manual:

I	=	Either signed two's complement integer in the range -32,768 to +32,767 or unsigned integer 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

In the instructions that utilize an effective address, the following coding conventions are used:

The indirect bit is set to 1 by coding the symbol @ anywhere in the effective address operand string.

The index bits are set by coding a comma followed by one of the digits 0-3 as the last operand of the operand string. The character "period" (.) can be used to set the index bits to 01. "Period" can be read to mean "address of the instruction". When the period is used, it is followed by either a plus or a minus sign followed by the displacement e.g., ".+7", or ".-2".

Note that setting the index bits to 01 by using the period is not the same as setting the index bits to 01 by coding a comma followed by a 1 when the instruction being coded is an extended class instruction. In the first case, the period is read by the assembler to mean the address of the instruction, so the assembler subtracts 1 from the coded displacement to allow for the way in which the CPU handles extended address calculations. In the second case, the assembler places the coded displacement in the assembled instruction without modification. For example, EJMP .+3 is not equivalent to EJMP 3,1. EJMP .+3 is equivalent to EJMP 2,1.

The displacement is coded as a signed number in the current assembler radix. This radix is the numbering system in which the program supplies numbers to the assembler. The default radix is base 8 or octal. The assembler radix can be changed by using the .RDX statement.

The assembler available with the ECLIPSE line of computers allows the programmer to place labels on instructions or locations in memory. When the assembler comes upon a label in the operand string of an effective address instruction, it automatically sets the index and displacement bits to the correct values. For a detailed discussion of the features and operation of the ECLIPSE line assembler, see the assembler manual (DGC 093-000017).

The fixed point and logical instructions which use the two accumulator-multiple operation format have several options that can be obtained by appending suffixes to the instruction mnemonic and by coding optional operands in the operand string. The characters to be coded are given below with their results.

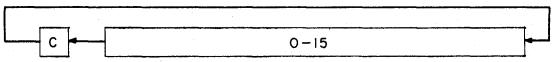
The characters in the column titled "class abbreviation" refer to specific fields in the two accumulator-multiple operation format. The characters in the column titled "coded character" show the various characters which may be coded for this option. The numbers in the column titled "result bits" show the bit settings in these fields resulting from each coded character. The comments in the column titled "operation" describe the effect of these bit settings.

CLASS ABBREVIATION	CODED CHARACTER	RESULT BITS	OPERATION
C	(option omitted)	00	Do not initialize the carry bit.
	Z	01	Initialize the carry bit to 0.
	O	10	Initialize the carry bit to 1.
	C	11	Initialize the carry bit to the complement of its present value.
SH	(option omitted)	00	Leave the result of the arithmetic or logical operation unaffected.
	L	01	Combine the carry and the 16-bit result into a 17-bit number and rotate it one bit left.
	R	10	Combine the carry and the 16-bit result into a 17-bit number and rotate it one bit right.
	S	11	Exchange the two 8-bit halves of the 16-bit result without affecting the carry.
#	(option omitted)	0	Load the result of the shift operation into ACD.
	#	1	Do not load the result of the shift operation into ACD.

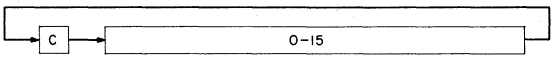
The following diagrams illustrate the operation of the shifter.

Coded Character — Shifter Operation

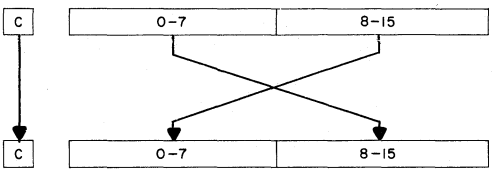
L — Left rotate one place. Bit 0 is rotated into the carry position, the carry bit into bit 15.



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.



The following operands initiate operations that test the result of the shift operation. If the tested condition is true, the next sequential instruction is skipped.

CLASS ABBREVIATION	CODED CHARACTER	RESULT BITS	OPERATION
SKIP	(option omitted)	000	Never skip.
	SKP	001	Always skip.
	SZC	010	Skip if carry = 0.
	SNC	011	Skip if carry ≠ 0.
	SZR	100	Skip if result = 0.
	SNR	101	Skip if result ≠ 0.
	SEZ	110	Skip if either carry or result = 0.
	SBN	111	Skip if both carry and result ≠ 0.

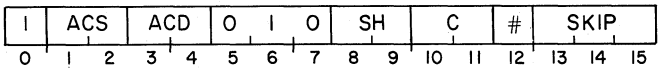
Instructions in the Two Accumulator-Multiple Operation format must not have both the "No Load" and the "Never Skip" options specified at the same time. These bit combinations are used by other instructions in the instruction set.

As an example of how to use these tables, assume that accumulator 3 contains a signed, two's complement number. Now consider the problem of determining whether this number is positive or negative. One way to determine this would be to place the number zero in another accumulator and use the SKIP IF ACS GREATER THAN ACD instruction, but this requires an extra instruction and also destroys the previous contents of the other accumulator. Another way to determine the sign of the number in accumulator 3 is to use the MOVE instruction and the power of the two accumulator-multiple operation format. With the MOVE instruction, the contents of AC3 can be placed in the shifter and shifted one bit to the left. This places the sign bit in the carry bit. The carry bit can then be tested for zero. In order to preserve the number in AC3, the instruction can prevent the output of the shifter from being loaded back into AC3.

The general form of the MOVE instruction is:

MOV<u><c></u><u><sh></u><u><#></u> <u>acs</u>,<u>acd</u><,<u>skip</u>>

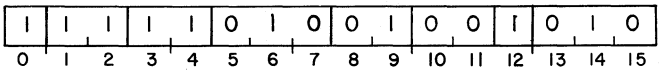
The general bit pattern of the MOVE instruction is:



To shift the number in AC3 one bit left without destroying the number, and skip the next sequential instruction if the bit shifted into the carry bit is zero, the following instruction could be coded:

MOVL# 3,3,SZC

This instruction would assemble into the following bit pattern:

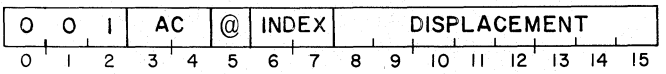


FIXED POINT ARITHMETIC

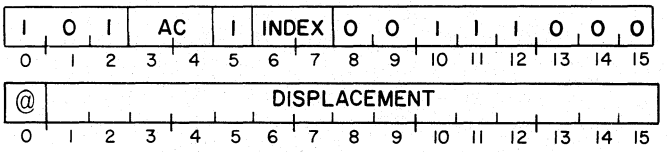
The fixed point instruction set performs binary arithmetic on operands in accumulators. The operands are 4, 16, or 32 bits in length and can be either signed or unsigned. The instruction set provides for loading, storing, adding, subtracting, multiplying, dividing, and comparing of fixed point operands.

LOAD ACCUMULATOR

LDA ac, <@>displacement<, index>



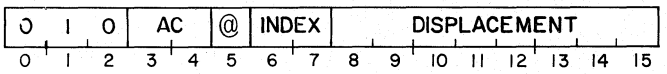
ELDA ac, <@>displacement<, index>



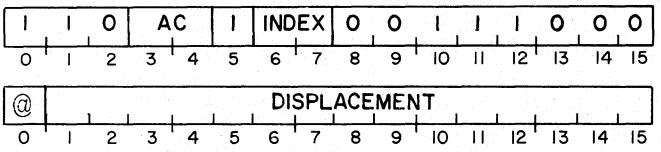
The word addressed by the effective address, "E", is placed in the specified accumulator. The previous contents of the AC are lost. The contents of the location addressed by "E" remain unchanged.

STORE ACCUMULATOR

STA ac, <@>displacement<, index>



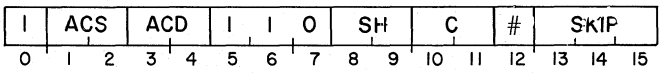
ESTA ac, <@>displacement<, index>



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.

ADD

ADD<c><sh><#> acs, acd<, skip>

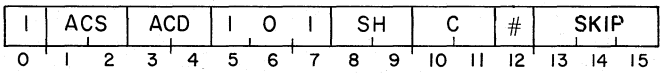


The carry bit is initialized to the specified value. The number in ACS is added to the number in ACD and the result is placed in the shifter. If the addition produces a carry of 1 out of the high-order bit, the carry bit is complemented. The specified shift operation is performed and the result of the shift is placed 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 $65,535_{10}$, the carry bit is complemented.

SUBTRACT

SUB<c><sh><#> acs, acd<, skip>

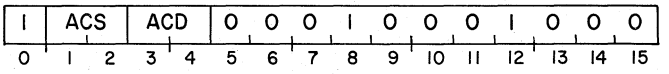


The carry bit is initialized to its specified value. The number in ACS is subtracted from the number in ACD by taking the two's complement of the number in ACS and adding it to the number in ACD. The result of the addition is placed in the shifter. If the operation produces a carry of 1 out of the high-order bit, the carry bit is complemented. The specified shift operation is performed and the result of the shift is placed in 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 or equal to the number in ACD the carry bit is complemented.

DECIMAL ADD

DAD acs, acd



The decimal digit contained in ACS bits 12-15 is added to the decimal digit contained in ACD bits 12-15. The carry bit is added to this result. The decimal units' position of the final result is placed in ACD bits 12-15 and the decimal carry is placed in the carry bit. The contents of ACS and bits 0-11 of ACD remain unchanged.

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.

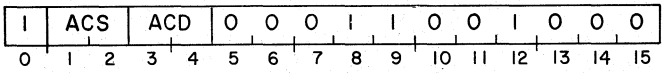
Example:

Assume that bits 12-15 of AC2 contain 9; bits 12-15 of AC3 contain 7; and the carry bit is 0, indicating no carry from the previous DECIMAL ADD. 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.

	BEFORE	AFTER
AC2	0 000 000 000 001 001	0 000 000 000 001 001
AC3	0 000 000 000 000 111	0 000 000 000 000 110
carry bit	0	1

DECIMAL SUBTRACT

DSB acs,acd



The decimal digit contained in ACS bits 12-15 is subtracted from the decimal digit contained in ACD bits 12-15. The complement of the carry bit is subtracted from this result. The decimal units' position of the final result is placed in ACD bits 12-15 and the complement of the decimal borrow is placed in the carry bit. In other words, if the final result is negative, a borrow is indicated, and the carry bit is set to 0. If the final result is positive, no borrow is indicated and the carry bit is set to 1.

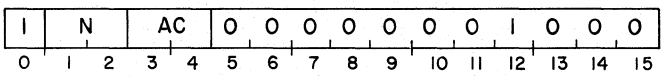
Example:

Assume that bits 12-15 of AC2 contain 9; bits 12-15 of AC3 contain 7; and the carry bit is 0, indicating a borrow from the previous DECIMAL SUBTRACT. After the instruction DSB 3,2 is executed, AC3 remains the same; bits 12-15 of AC2 contain 1; and the carry bit is set to 1, indicating no borrow from this DECIMAL SUBTRACT.

	BEFORE	AFTER
AC2	0 000 000 000 001 001	0 000 000 000 000 001
AC3	0 000 000 000 000 111	0 000 000 000 000 111
carry bit	0	1

ADD IMMEDIATE

ADI n,ac



The contents of the immediate field "N", plus 1, are added to the unsigned, 16-bit number contained in AC and the result is placed in AC. The carry bit remains unchanged.

NOTE The assembler takes the coded value of "n" and subtracts one from it before placing it in the immediate field. Therefore, the programmer should code the exact value that he wishes to add.

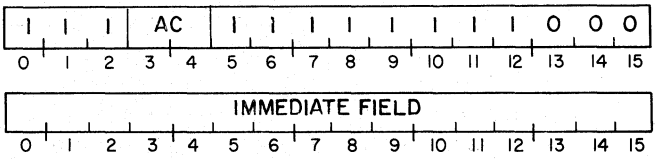
Example:

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

	BEFORE	AFTER
AC2	1 111 111 111 111 101	0 000 000 000 000 001
carry bit	either 0 or 1	unchanged

EXTENDED ADD IMMEDIATE

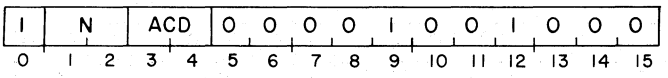
ADDI i,ac



The contents of the immediate field are treated as a signed, two's complement number and added to the signed, two's complement number contained in AC and the result is placed in AC. The carry bit remains unchanged.

SUBTRACT IMMEDIATE

SBI n,acd



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

> The assembler takes the coded value of "n" and subtracts one from it before placing it in the immediate field. Therefore, the programmer should code the exact value that he wishes to subtract.

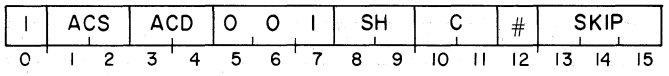
Example:

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

	BEFORE	AFTER
AC2	0 000 000 000 000 011	1 111 111 111 111 111
carry bit	either 0 or 1	unchanged

NEGATE

NEG<c><sh><#> acs,acd<,skip>

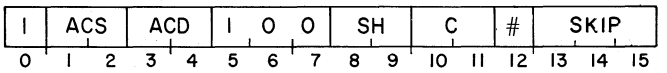


The carry bit is initialized to the specified value. The two's complement of the number in ACS is placed in the shifter. If the negate operation produces a carry of 1 out of the high-order bit, the carry bit is complemented. The specified shift operation is performed and the result is placed in ACD if the no-load bit is 0. If the skip condition is true, the next sequential word is skipped.

NOTE If ACS contains 0, the carry bit is complemented.

ADD COMPLEMENT

ADC<c><sh><#> acs,acd<,skip>

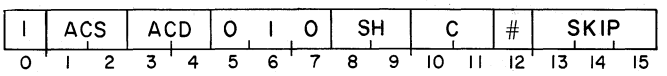


The carry bit is initialized to the specified value. The logical complement of the number in ACS is added to the number in ACD and the result is placed in the shifter. If the addition produces a carry of 1 out of the high-order bit, the carry bit is complemented. The specified shift operation is performed, and the result of the shift is loaded 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 carry bit is complemented.

MOVE

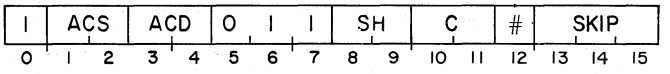
MOV<c><sh><#> acs,acd<,skip>



The carry bit is initialized to the specified value. The contents of ACS are placed in the shifter. The specified shift operation is performed and the result of the shift is loaded into ACD if the no-load bit is 0. If the skip condition is true, the next sequential word is skipped.

INCREMENT

INC<c><sh><#> acs,acd<,skip>

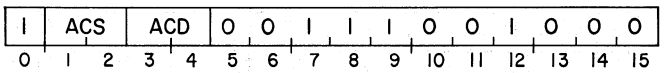


The carry bit is initialized to the specified value. The number in ACS is incremented by one and the result is placed in the shifter. If the incrementation produces a carry of 1 out of the high-order bit, the carry is complemented. The specified shift operation is performed, and the result of the shift is loaded 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 carry bit is complemented.

EXCHANGE ACCUMULATORS

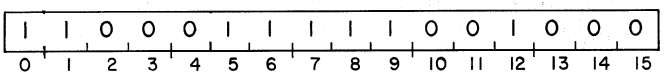
XCH acs,acd



The original contents of ACS are placed in ACD and the original contents of ACD are placed in ACS.

UNSIGNED MULTIPLY

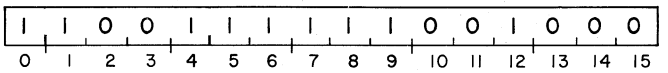
MUL



The 16-bit unsigned number in AC1 is multiplied by the 16-bit unsigned number in AC2 to yield a 32-bit unsigned intermediate result. The 16-bit unsigned number in AC0 is added to the intermediate result to produce the final result. The final result is a 32-bit unsigned 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

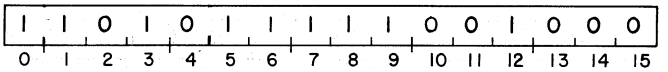
MULS



The 16-bit signed two's complement number in AC1 is multiplied by the 16-bit signed two's complement number in AC2 to yield a 32-bit signed two's complement intermediate result. The 16-bit signed two's complement number in AC0 is added to the intermediate result to produce the final result. The final result is a 32-bit signed 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.

UNSIGNED DIVIDE

DIV

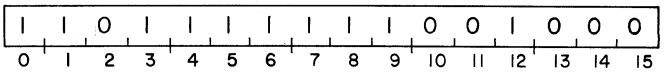


The 32-bit unsigned number contained in AC0 and AC1 is divided by the 16-bit unsigned number in AC2. The quotient and remainder are 16-bit unsigned numbers and are placed in AC1 and AC0, respectively. The carry bit is set to 0. The contents of AC2 remain unchanged.

NOTE Before the divide operation takes place, AC0 is compared to AC2. If the number in AC0 is greater than or equal to the number in AC2, an overflow condition is indicated. The carry bit is set to 1, and the operation is terminated. All operands remain unchanged.

SIGNED DIVIDE

DIVS



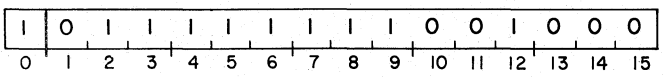
The 32-bit signed two's complement number contained in AC0 and AC1 is divided by the 16-bit signed two's complement number in AC2. The quotient and remainder are 16-bit signed numbers and occupy AC1 and AC0, respectively. The sign of the quotient is determined by the rules of algebra. The sign of the remainder is always the same as the sign of the dividend, except that a zero quotient or a zero remainder is always positive.

The carry bit 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. The carry bit is set to 1, and the operation is terminated. The contents of AC0 and AC1 are unpredictable.

SIGN EXTEND AND DIVIDE

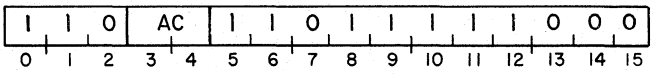
DIVX



The sign of the number in AC1 is extended into AC0 by placing a copy of bit 0 of AC1 in each bit of AC0. After the sign extension, a SIGNED DIVIDE is performed.

HALVE

HLV ac



The signed two's complement number contained in AC is divided by 2 and rounded toward 0. The result is placed in AC.

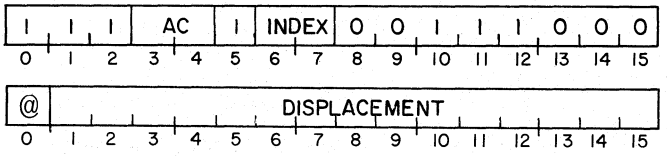
If the number is positive, division is accomplished by shifting the number right one bit. If the number is negative, division is accomplished by negating the number, shifting it right one bit, and negating it again.

LOGICAL OPERATIONS

The logical instruction set performs logical operations on operands in accumulators. The operands are 16 bits long and are treated as unstructured binary quantities. The logical operations included in this set are: AND, inclusive OR, exclusive OR, AND with complemented source, and COMPLEMENT. The logical instruction set also provides instructions for shifting operands in accumulators. Single length (16 bits) or double length (32 bits -- formed by combining two adjoining accumulators) operands can be logically shifted left or right in one or four bit increments. The four bit increments are called hexadecimal or "hex" digits.

LOAD EFFECTIVE ADDRESS

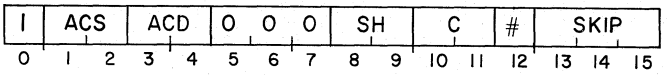
ELEF ac,<@>displacement<,index>



The effective address "E" is computed and placed in bits 1-15 of AC. Bit 0 of AC is set to 0. The previous contents of AC are lost.

COMPLEMENT

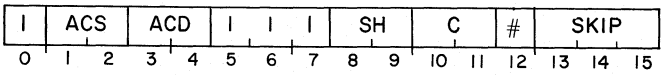
COM<c><sh><#> acs,acd<,skip>



The carry bit is initialized to the specified value. The logical complement of the number in ACS is placed in the shifter. The specified shift operation is performed and the result is placed in ACD if the no-load bit is 0. If the skip condition is true, the next sequential word is skipped.

AND

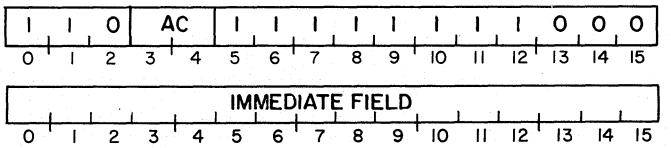
AND<c><sh><#> acs,acd<,skip>



The carry bit is initialized to the specified value. The logical AND of ACS and ACD is placed in the shifter. Each bit placed in the shifter is 1 only if the corresponding bit in both ACS and ACD is one; otherwise the result bit is 0. The specified shift operation is performed and the result is placed in ACD if the no-load bit is 0. If the skip condition is true, the next sequential word is skipped.

AND IMMEDIATE

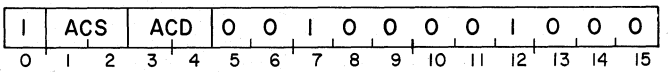
ANDI i,ac



The contents of the immediate field are treated as an unstructured 16-bit quantity. The logical AND of the contents of the immediate field and the contents of AC is placed in AC.

INCLUSIVE OR

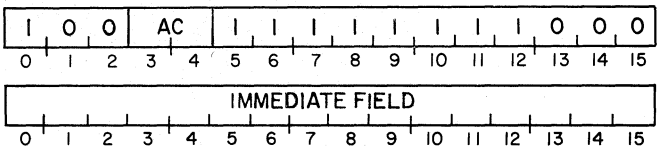
IOR acs,acd



The contents of ACS are inclusively OR'd with the contents of ACD and the result is placed in ACD. A bit position in the result is set to 1 if the corresponding bit position in one or both operands contains a 1; otherwise, the result bit is set to 0. The contents of ACS remain unchanged.

INCLUSIVE OR IMMEDIATE

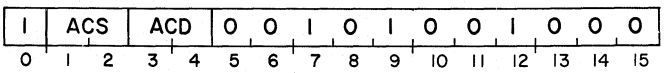
IORI i,ac



The contents of the immediate field are treated as an unstructured 16-bit quantity. The logical inclusive OR of the contents of the immediate field and the contents of AC is placed in AC.

EXCLUSIVE OR

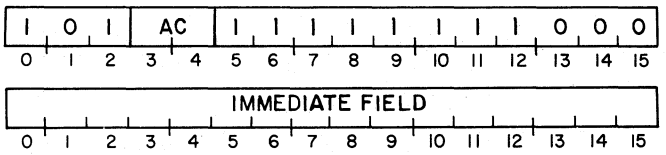
XOR acs,acd



The contents of ACS are exclusively OR'd with the contents of ACD and the result is placed in ACD. A bit position in the result is set to 1 if the corresponding bit positions in the two operands are unlike; otherwise, the result bit is set to 0. The contents of ACS remain unchanged.

EXCLUSIVE OR IMMEDIATE

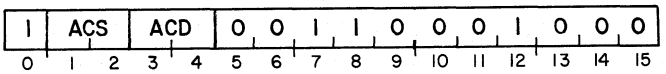
XORI i,ac



The contents of the immediate field are treated as an unstructured 16-bit quantity. The logical exclusive OR of the contents of the immediate field and the contents of AC is placed in AC.

AND WITH COMPLEMENTED SOURCE

ANC acs,acd



The logical complement of the contents of ACS is AND'd with the contents of ACD and the result is placed in ACD. A bit position in the result is set to 1 if the corresponding bit positions in ACS and ACD contain a 0 and 1, respectively; otherwise, the result bit is set to zero. The contents of ACS remain unchanged.

Example:

The AND WITH COMPLEMENTED SOURCE instruction can be used to reset bits through a mask. If the operand in ACD contains bit positions that were set to 1 through a mask with the INCLUSIVE OR instruction, the AND WITH COMPLEMENTED SOURCE instruction will set those bits to 0 using the same mask.

Assume that AC0 contains 0, AC1 contains 010357_8 and AC2 contains 170441_8. After the instruction IOR 1,0 is executed, AC0 contains 010357_8.

	BEFORE	AFTER
AC0	0 000 000 000 000 000	0 001 000 011 101 111
AC1	0 001 000 011 101 111	0 001 000 011 101 111

After the instruction IOR 2,0 is executed AC0 contains 170757_8.

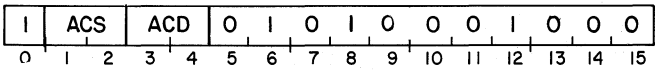
	BEFORE	AFTER
AC0	0 001 000 011 101 111	1 111 000 111 101 111
AC2	1 111 000 100 100 001	1 111 000 100 100 001

If it is desired to set to 0 all those bits that were set to 1 by the first INCLUSIVE OR instruction, the AND WITH COMPLEMENTED SOURCE instruction will do it. After the instruction ANC 1,0 is executed, AC0 contains 160400_8.

	BEFORE	AFTER
AC0	1 111 000 111 101 111	1 110 000 100 000 000
AC1	0 001 000 011 101 111	0 001 000 011 101 111

LOGICAL SHIFT

LSH acs,acd



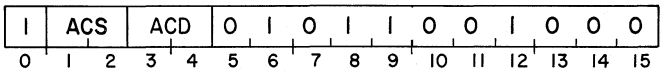
The contents of ACD are shifted left or right depending on the number contained in bits 8-15 of ACS. The 8-bit signed two's complement number contained in bits 8-15 of ACS determines the direction of the shift and the number of bits to be shifted. If the number in bits 8-15 of ACS is positive, shifting 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_{10}, all bits of ACD are set to 0. The carry bit and the contents of ACS remain unchanged.

DOUBLE LOGICAL SHIFT

DLSH acs,acd



The 32-bit number contained in ACD and ACD+1 is shifted left or right depending on the number contained in bits 8-15 of ACS. The 8-bit signed 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.

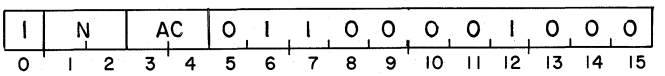
The number of bits shifted is equal to the magnitude of the number in bits 8-15 of ACS. Bits shifted out are lost, and the vacated bit positions are filled with zeroes. The carry bit and the contents of ACS remain unchanged.

NOTES 1. If the magnitude of the number in bits 8-15 of ACS is greater than 31_{10}, all bits of ACD and ACD+1 are set to 0.

2. If ACD is specified as AC3, then ACD+1 is AC0.

HEX SHIFT LEFT

HXL n,ac

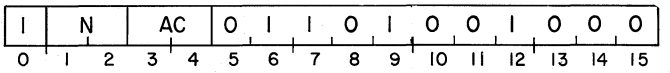


The contents of AC are shifted left a number of hex digits depending upon the immediate field "N". The number of digits shifted is equal to N+1. Bits shifted out are lost, and the vacated bit positions are filled with zeroes. If N is equal to 3, then all 16 bits of AC are shifted out and all bits of 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, the programmer should code the exact number of hex digits that he wishes to shift.

HEX SHIFT RIGHT

HXR n,ac

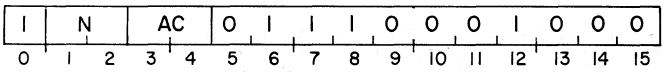


The contents of AC are shifted right a number of hex digits depending upon the immediate field, "N". The number of digits shifted is equal to N+1. Bits shifted out are lost, and the vacated bit positions are filled with zeroes. If N is equal to 3, then all 16 bits of AC are shifted out and all bits of 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, the programmer should code the exact number of hex digits that he wishes to shift.

DOUBLE HEX SHIFT LEFT

DHXL n,ac



The 32-bit number contained in AC and AC+1 is shifted 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.

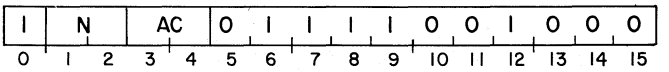
NOTES 1. If AC is specified as AC3, then AC+1 is AC0.

2. 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.

3. 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



The 32-bit number contained in AC and AC+1 is shifted 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.

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

2. 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.

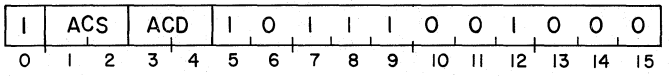
3. If N is equal to 3, the contents of AC are placed in AC+1 and AC is filled with zeroes.

BYTE MANIPULATION

In addition to performing operations on structured and unstructured 16-bit quantities, the instruction set of the ECLIPSE line of computers allows the loading and storing of 8-bit bytes. The LOAD BYTE and STORE BYTE instructions can be used with the SWAP option of the two accumulator-multiple operation instructions and with the hex shift instructions to perform character operations.

LOAD BYTE

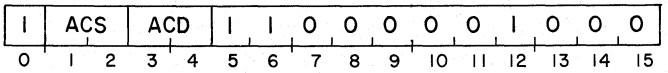
LDB acs,acd



The 8-bit byte addressed by the byte pointer contained in ACS is placed in bits 8-15 of ACD. Bits 0-7 of ACD are set to 0. The contents of ACS remain unchanged.

STORE BYTE

STB acs,acd



Bits 8-15 of ACD are placed in the byte addressed by the byte pointer contained in ACS. The contents of ACS and ACD remain unchanged.

Example:

The following instruction sequence will convert an unsigned integer to its corresponding octal representation and place the result in six bytes in memory. AC0 contains the integer, AC3 contains a byte pointer to the low-order byte of the destination field.

When the routine is finished, and control is transferred to the location "OUT", the integer has been converted to octal and the byte pointer in AC3 points to the high-order byte of the result.

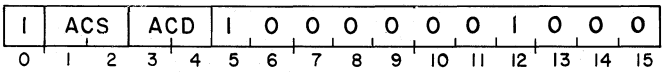
```
        LDA     2,CON6  ;GET COUNT
        STA     2,CNT   ;STORE IT
        LDA     2,MASK  ;GET MASK FOR CHAR
                        ; AND SHIFT COUNT
LOOP:   DLSH    2,0     ;SHIFT ONE OCTAL
                        ; DIGIT
        HXR     1,1     ;SHIFT AC1 4 BITS
                        ; RIGHT
        MOVZR   1,1     ;SHIFT AC1 1 BIT
                        ; RIGHT
        IOR     2,1     ;OR IN BITS FOR
                        ; CHARACTER
        MOVS    1,1     ;PUT CHAR IN LOW-
                        ; ORDER BYTE
        STB     3,1     ;STORE BYTE
        DSZ     CNT     ;DONE?
        JMP     .+2     ;NO
        JMP     OUT     ;YES
        SBI     1,3     ;DECREMENT AC3 BY 1
        JMP     LOOP    ;DO NEXT DIGIT
CON6:   6
CNT:    0
MASK:   030375          ;CHAR MASK IN HI
                        ; BYTE, SHIFT COUNT
                        ; IN LOW BYTE
OUT:    ...
        ...
        ...
```

BIT MANIPULATION

In addition to performing operations on structured and unstructured 16 bit quantities and on 8-bit bytes, the standard instruction set allows operations to be performed on individual bits in accumulators and in memory. This set of instructions includes operations that locate leading bits, set bits, and test bits.

SET BIT TO ONE

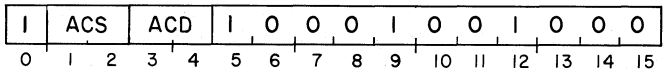
BTO acs,acd



A 32-bit bit pointer is formed 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 accumulator contents are treated as the low-order 16-bits of the bit pointer and the high-order 16 bits are assumed to be 0. The addressed bit in memory is set to 1. The contents of ACS and ACD remain unchanged.

SET BIT TO ZERO

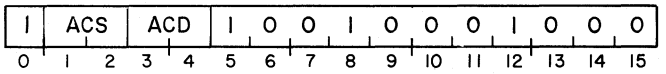
BTZ acs,acd



A 32-bit bit pointer is formed 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 accumulator contents are treated as the low-order 16 bits of the bit pointer and the high-order 16 bits are assumed to be 0. The addressed bit in memory is set to 0. The contents of ACS and ACD remain unchanged.

SKIP ON ZERO BIT

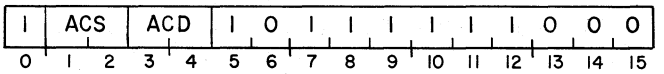
SZB acs,acd



A 32-bit bit pointer is formed 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 accumulator contents are treated as the low-order 16 bits of the bit pointer and the high-order 16 bits are assumed to be 0. If the addressed bit in memory is 0, the next sequential word is skipped. The contents of ACS and ACD remain unchanged.

SKIP ON NON-ZERO BIT

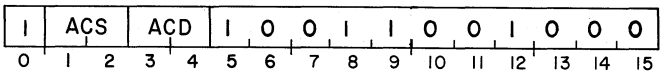
SNB acs,acd



A 32-bit bit pointer is formed 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 accumulator contents are treated as the low-order 16 bits of the bit pointer and the high-order 16 bits are assumed to be 0. If the addressed bit in memory is 1, the next sequential word is skipped. The contents of ACS and ACD remain unchanged.

SKIP ON ZERO BIT AND SET TO ONE

SZBO acs,acd

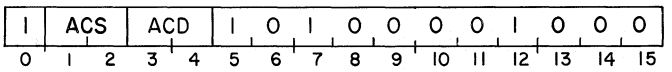


A 32-bit bit pointer is formed from the contents of ACS and ACD. ACS contains the high-order 16 bits and ACD contains the low-order 16 bits of the bit pointer. If ACS and ACD are specified as the same accumulator, the accumulator contents are treated as the low-order 16 bits of the bit pointer and the high-order 16 bits are assumed to be 0. The addressed bit in memory is set to 1. If the bit was 0 before being set to 1, the next sequential word is skipped. The contents of ACS and ACD remain unchanged.

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.

LOCATE LEAD BIT

LOB acs,acd

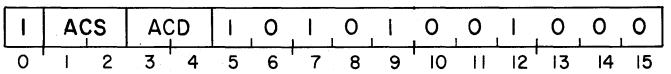


The contents of ACS are inspected for high-order zeroes. A number equal to the number of high-order zeroes is added to the 16-bit signed two's complement number contained in ACD. The contents of ACS and the state of the carry bit remain unchanged.

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

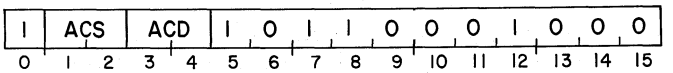


The contents of ACS are inspected for high-order zeroes. A number equal to the number of high-order zeroes is added to the 16-bit signed two's complement number contained in ACD. The leading 1 in ACS is set to 0. The state of the carry bit remains unchanged.

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

COUNT BITS

COB acs,acd



The contents of ACS are inspected for 1's. A number equal to the number of 1's in ACS is added to the 16-bit signed two's complement number contained in ACD. The contents of ACS and the state of the carry bit remain unchanged.

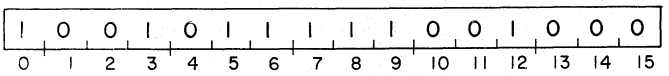
NOTE If ACS and ACD are specified as the same accumulator, the instruction functions as described above, except that since ACS and ACD are the same accumulator, the contents of ACS will be changed.

DATA MOVEMENT

Two instructions are provided in the ECLIPSE line for the rapid, convenient movement of data from one location in memory to another. These instructions move from 1 to 32,768 words in one operation. The BLOCK ADD AND MOVE instruction also adds a constant to each word as it is moved. This feature allows easy relocation of address constants.

BLOCK ADD AND MOVE

BAM



Words are moved sequentially from one memory location to another. The words moved are treated as 16-bit unsigned integers. After a word has been fetched from the source location, and before it is stored at the destination location, the 16-bit unsigned integer contained in AC0 is added to it. If the addition produces a carry of 1 out of the high-order bit, no indication is given.

The address of the source location is contained in bits 1-15 of AC2. The address of the destination location is contained in bits 1-15 of AC3. If bit 0 of either AC2 and AC3 is 1, it is assumed that the address contained in bits 1-15 is an indirect address. Before the data movement occurs, the indirection chain is followed and the resultant effective address is placed in the accumulator.

The number of words moved is equal to the 16-bit unsigned number contained in AC1. This number must be greater than 0 and less than or equal to 100000_8. If the number contained in AC1 is outside these bounds, no data is moved and the contents of the accumulators is unchanged.

ACCUMULATOR DESCRIPTIONS

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.

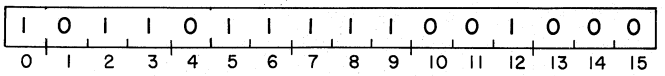
Words are moved in consecutive, ascending order according to their addresses. The next address after 77777_8 is 0 for both fields. The fields may overlap in any way.

NOTE Due to the potentially long time that may be required to perform this instruction in relation to I/O requests, this instruction is interruptable. 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 BLOCK ADD AND MOVE instruction. Because the addresses and the word count are updated after every word stored, any interrupt service routine that returns control to the interrupted program via the address stored in memory location 0 will correctly restart the BLOCK ADD AND MOVE instruction.

When updating the source and destination addresses, the BLOCK ADD AND MOVE instruction forces bit 0 of the result to 0. This ensures that upon return from an interrupt, the BLOCK ADD AND MOVE instruction will not try to resolve an indirect address in either AC2 or AC3.

BLOCK MOVE

BLM



The BLOCK MOVE instruction is equivalent to 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 interruptable in the same manner as the BLOCK ADD AND MOVE instruction. The note for BLOCK ADD AND MOVE also applies to BLOCK MOVE.

Example:

The following sequence of instructions will create a 17 word table of constants. The first word in the table will have the value 0, the second word will have the value 1, and so on. The last word in the table will have the value 16_{10}.

```
        LDA     2,TBLAD     ;PUT ADDRESS OF
                            ; TABLE IN AC2
        INC     2,3         ;PUT ADDRESS OF
                            ; TABLE+1 IN AC3
        SUBO    0,0         ;SET AC0 to 0
        STA     0,0,2       ;SET FIRST WORD
                            ; OF TABLE TO 0
        INC     0,0         ;AC0 = ADDEND OF 1
        MOV     0,1         ;PUT 16(10)
        HXL     1,1         ; IN AC1
        BAM                 ;CREATE TABLE
        JMP     TABLE+21    ;JUMP AROUND
                            ; TABLE--17(10) = 21(8)
TBLAD:  TABLE               ;ADDRESS OF TABLE
TABLE:  .BLK    21          ;RESERVE 21(8) WORDS
                            ; FOR TABLE
```

The first word moved is moved from TABLE+0 to TABLE+1. As it is moved, it is incremented by 1. The second word moved is moved from TABLE+1 to TABLE+2. As it is moved, it is incremented by 1. The moving and adding continues until the table is filled.

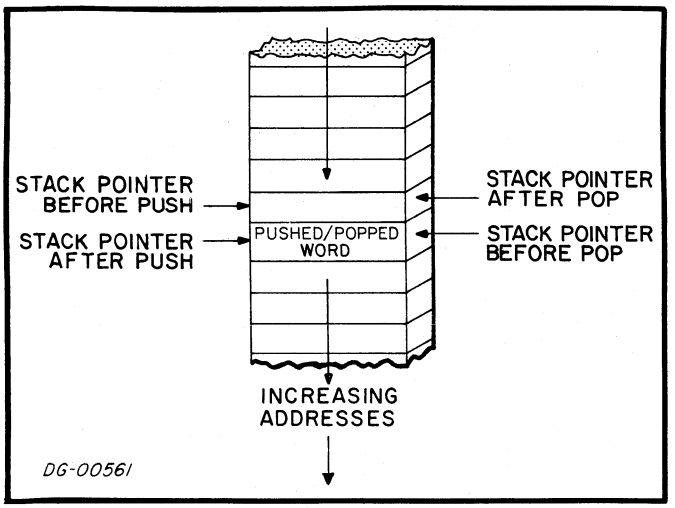
STACK MANIPULATION

An important feature of the ECLIPSE line of computers is the stack manipulation facility. A Last-In/First-Out (LIFO) or "Push-Down" stack is maintained by the processor. The stack facility provides an expandable area of temporary storage for variables, data, return addresses, subroutine arguments, etc. An important byproduct of the stack facility is that storage locations are reserved only when needed. When a procedure is finished with its portion of the stack, those memory locations are reclaimed and are available for use by some other procedure.

The operation of the stack depends upon the contents of some reserved storage locations. Locations 40-43$_8$ are the stack control words for the stack. The locations and their contents are described below.

Stack Pointer

The stack pointer is contained in location 40$_8$. The stack pointer is the address of the "top" of the stack and is affected by operations that either "push" objects onto or "pop" objects off of the stack. A push operation increments the stack pointer by 1 and then places the "pushed" object in the word addressed by the new value of the stack pointer. A pop operation takes the word addressed by the current value of the stack pointer and places it in some new location and then decrements the stack pointer by 1.



Frame Pointer

The frame pointer is contained in location 41$_8$. The frame pointer is used to reference an area in the user stack called a "frame". A frame is that portion of the stack which is reserved for use by a certain procedure. The frame pointer usually points to the first available word minus 1 in the current frame. The frame pointer is also used by the RETURN instruction to reset the user stack pointer.

Stack Limit

The stack limit is contained in location 42$_8$. The stack limit is the address that is used to determine the presence of a stack overflow condition. After any stack operation that pushes objects onto the stack, the stack pointer is compared to the stack limit. If the stack pointer is greater than the stack limit, a stack overflow condition is indicated and a stack fault occurs.

Stack Fault Address

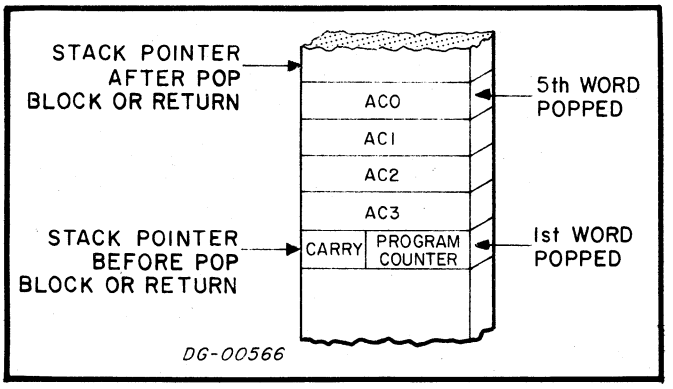
The stack fault address is contained in location 43$_8$. The stack fault location contains the address of the stack fault handler. Bit 0 of the stack fault location may be set to 1, indicating that the address contained in bits 1-15 is an indirect address.

Return Block

A return block is defined as a block of five words that is pushed onto the stack in order to allow convenient return to the calling program. The contents of the return block may vary slightly depending upon which instruction pushes the block, but the purpose of the block is always the same -- to allow orderly return by the POP BLOCK instruction, the RETURN instruction, or the RESTORE instruction. The format of the return block, therefore, is determined by how it is used in the return sequence. The format of the return block is as follows:

WORD # POPPED	DESTINATION
1	Bit 0 placed in the carry bit. Bits 1-15 placed in the program counter
2	AC3
3	AC2
4	AC1
5	AC0

In the stack, the return block looks like this:



Stack Frames

In order to implement re-entrant subroutines, a new area of temporary storage must be available for each execution of a called subroutine. The easiest way to accomplish this is for the subroutine to use the stack for temporary storage. A "stack frame" is defined as that portion of the stack which is available to the called routine. In general, the stack frame belonging to a subroutine begins with the first word in the stack after the return block pushed by the called routine and contains all words in the stack up to, and including, the return block pushed by any routine which the called routine calls. Variables and arguments can be transmitted from the calling routine to the called routine by placing them in prearranged positions in the calling routine's stack frame. Because the SAVE instruction sets the frame pointer to the last word in the return block, these variables and arguments can be referenced by the called program as a negative displacement from the frame pointer. The called routine should ensure that reference to the calling routine's stack frame is made only with the permission of the calling routine.

Stack Protection

Two types of protection are available for users of the stack. The two conditions that can be detected as error conditions are stack "overflow" and stack "underflow". Stack overflow occurs when a program pushes data into the area beyond that allocated for the stack. If stack overflow is allowed to occur, data will be pushed into areas that are reserved for other purposes, and information or instructions may be destroyed. Stack underflow occurs when a program pops data from the area below the area allocated for the stack. If stack underflow is allowed to occur, the program will be operating with information that will lead it to an incorrect conclusion. In addition, it is possible that the program will push data in the underflow area, destroying either data or instructions. Both underflow and overflow protection can be enabled and disabled by the program.

To enable underflow protection, the area allocated for 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 is indicated and a stack fault occurs.

Underflow protection can be disabled in two ways. The first way is to allocate space for the stack starting at a location greater than 401_8. In this way, an underflow stack fault will not occur unless the program underflows the stack and continues to pop objects off the stack until the stack pointer is less than 400_8. The second way to disable underflow protection is to 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 without interference from the stack underflow mechanism.

To make stack overflow protection meaningful, the stack limit must be initialized to the address of the last word allocated for the stack minus at least 10_{10}. The reason for this is that stack overflow is sensed only at the end of a push operation. It is possible to push a 5-word return block starting at the address contained in the stack limit. Stack overflow will not be sensed until the fifth word of the return block is pushed. After the fifth word is pushed, stack overflow will be indicated, and another 5-word return block is pushed by the stack overflow mechanism before control is transferred to the stack fault routine. This means that at least ten stack words must be allocated beyond the address set in the stack limit. If the state of the floating point feature is to be pushed, it is possible to push 23 words beyond the stack limit. For a VECTOR stack, it is possible to push 11 words past the stack limit before stack overflow is sensed.

To disable overflow protection, the stack limit may be set to 77777_8. This will ensure that a stack overflow condition is never indicated.

To disable both underflow and overflow protection, bit 0 of both the stack pointer and the stack limit should be set to 1. In addition bits 1-15 of the stack limit should be set to 77777_8. With the stack pointer and the stack limit set up in this way, all protection devices for the stack are disabled.

Stack Protection Faults

After every operation that pushes data onto the stack, a check is made for overflow protection. The stack pointer and the stack limit are treated as unsigned 16-bit integers and compared. If the stack pointer is greater than the stack limit, a stack overflow condition exists. If a stack overflow condition exists, the processor pushes a return block onto the stack with the program counter in the return block pointing to the next logical instruction after the stack instruction that caused the fault. Bit 0 of the stack pointer is set to 0 and bit 0 of the stack limit is set to 1. After the return block has been pushed and bit 0 of the stack pointer and the stack limit have been set, the processor executes a "jump indirect" to the stack fault address.

After every operation that pops data off the stack, a check is made for underflow protection. If the stack pointer is less than 400_8 and bit 0 of the stack limit is 0, a stack underflow condition exists. If a stack underflow condition exists, the processor sets the stack pointer equal to the stack limit and pushes a return block with the program counter in

the return block pointing to the instruction immediately after the stack instruction that caused the fault. Bit 0 of the stack pointer is set to 0 and bit 0 of the stack limit is set to 1. After the return block has been pushed and bit 0 of the stack pointer and the stack limit have been set, the processor executes a "jump indirect" to the stack fault address.

It is up to the stack fault handler to determine the exact nature of the stack error. This can be done by looking at the contents of the stack pointer and the stack limit. When the stack fault routine receives control, if the address contained in bits 1-15 of the stack pointer is not greater than the address in bits 1-15 of the stack limit by 5, then the error was a stack overflow error. If the address in bits 1-15 of the stack pointer is greater than the address in bits 1-15 of the stack limit by exactly 5, then the error was a stack underflow error. Once the stack fault routine has determined the nature of the error, it can take appropriate action, such as allocating more stack space or terminating the program.

Initialization of the Stack Control Words

Before the first operation on the stack can be performed, the stack control words must be initialized. The rules for initialization are as follows:

Stack Pointer

The stack pointer must be initialized to the beginning address of the stack area minus one. If stack underflow protection is desired, the stack pointer must be initialized to 400_8 and the stack area must start at 401_8. If stack underflow protection is not desired, start the stack at some location greater than 401_8. If it is desired to have all or a portion of the stack area in page zero, bit 0 of both the stack pointer and the stack limit must be set to 1.

Frame Pointer

If the main user program is going to use the frame pointer, it should be initialized to the same value as the stack pointer. Otherwise, the frame pointer can be initialized in a subroutine by the SAVE instruction.

Stack Limit

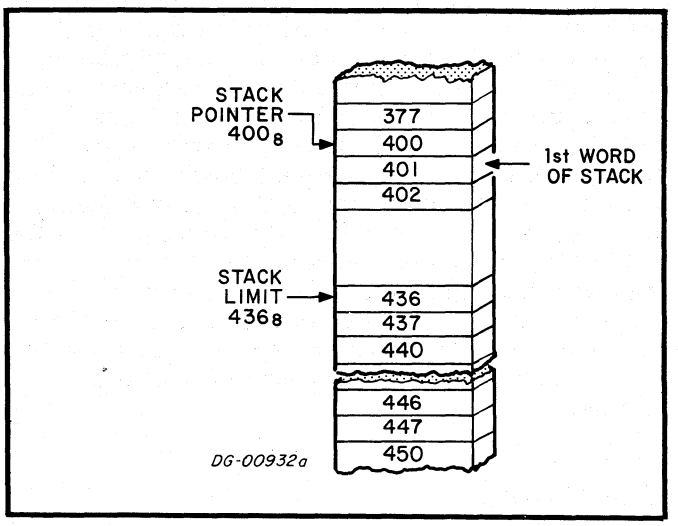
In order for stack operations to be meaningful, the stack limit must be initialized to a value greater than the stack pointer. If stack overflow protection is desired, the stack limit should be initialized to the last address allocated for the stack minus at least ten. If stack overflow protection is not desired, the stack limit should be initialized to 77777_8. If it is desired to have all or a portion of the stack area in page zero, bit 0 of both the stack pointer and the stack limit must be set to 1.

Stack Fault Address

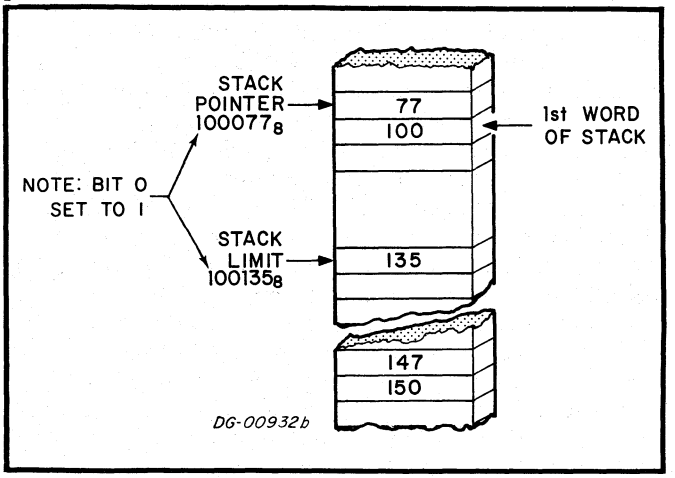
The stack fault address should be initialized to the address of the routine that is to receive control in the event of a stack overflow or underflow. Bit 0 may be set to 1 to indicate an indirect address.

Examples:

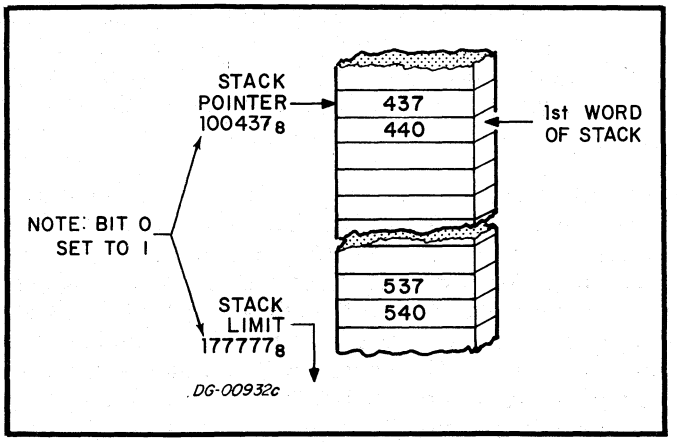
Stack area of 50_8 words with overflow and underflow protection



Stack area of 50_8 words in page zero with overflow protection



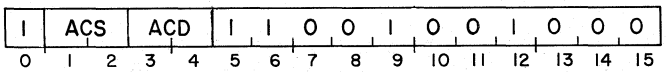
Stack area of 100_8 words with no protection



STACK MANIPULATION INSTRUCTIONS

PUSH MULTIPLE ACCCUMULATORS

PSH acs,acd

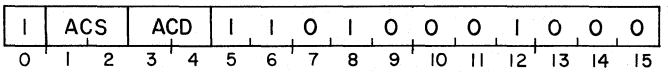


The set of accumulators starting with ACS and ending with ACD is pushed onto the stack. The accumulators are pushed in ascending order, starting with the AC specified by ACS and continuing up to and including the AC specified by ACD, with AC0 following AC3.

The contents of the accumulators remain unchanged. If ACS is equal to ACD, only ACS is pushed.

POP MULTIPLE ACCUMULATORS

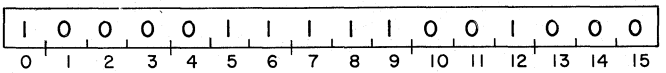
POP acs,acd



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

PUSH RETURN ADDRESS

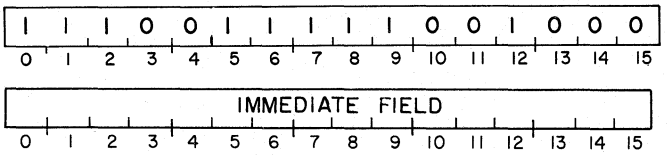
PSHR



Two is added to the present value of the program counter and the result is pushed onto the stack.

SAVE

SAVE i



A return block is pushed onto the stack. After the fifth word of the return block is pushed, the value of the stack pointer is placed in the frame pointer and in AC3. The format of the five words pushed 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 the return block has been pushed, bits 0-15 of the stack pointer are placed in bits 0-15 of the frame pointer.

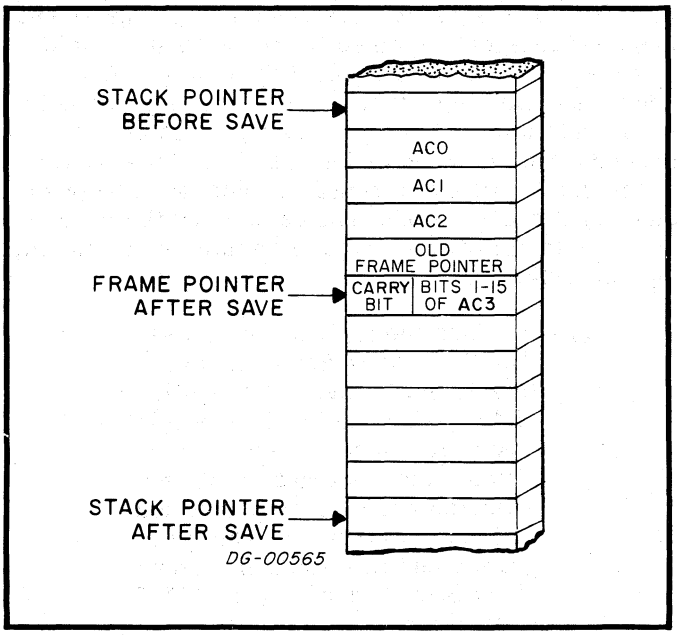
After the frame pointer has been set up, the 16-bit unsigned integer contained in the immediate field is added to the stack pointer. The integer that is added is called the "frame size". The purpose is to allocate a portion of the stack for use by the procedure which executed the SAVE. This portion of 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, and so forth.

NOTE Before the instruction is executed, a check for stack overflow is performed. If execution of the SAVE instruction would result in a stack overflow condition, the instruction is not executed and a stack protection fault is performed. The program counter in the fault return block is the address of the SAVE instruction.

Example:

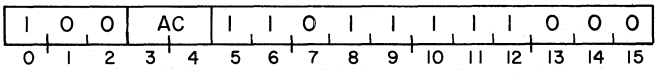
If a subroutine receives control via a JUMP TO SUBROUTINE instruction, then the following SAVE instruction will save all the return information, allocate a 6-word block in the stack for use by the procedure, and set the frame pointer to the address of the last word in the return block.

```
        JSR   LOOP
        ...
        ...
        ...
LOOP:   SAVE  6
        ...
        ...
        ...
```



MODIFY STACK POINTER

MSP ac



The contents of the specified AC are added to the contents of the stack pointer and the result is compared to the stack limit. If the result is greater than the stack limit, a stack protection fault is performed. The program counter in the fault return block is the address of the MODIFY STACK POINTER instruction. The stack pointer is left unchanged.

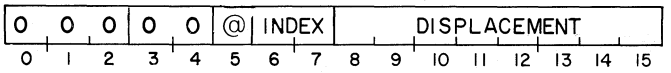
If the result is not greater than the stack limit, the result is placed in the stack pointer.

PROGRAM FLOW ALTERATION

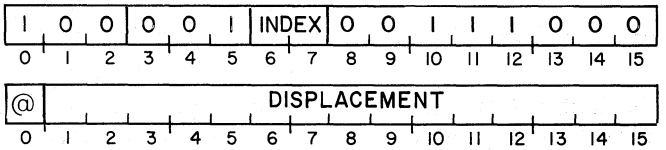
As stated previously, the normal method of program execution is sequential. That is, the processor will continue to retrieve instructions from sequentially addressed locations in memory until directed to do otherwise. Instructions are provided in the instruction set that alter this sequential flow. Program flow alteration is accomplished by placing a new value in the program counter. Sequential operations will then continue with the instruction addressed by this new value. Instructions are provided that change the value of the program counter, change the value of the program counter and save a return address, or modify a memory location by incrementing or decrementing and skip the next sequential instruction if the result is zero. In addition to these operations, there are also instructions that alter the program flow while saving or restoring the state of the machine. These instructions allow convenient implementation of nested subroutines, re-entrant routines, and recursive procedures.

JUMP

JMP <@>displacement<,index>



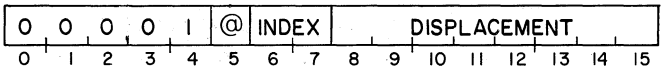
EJMP <@>displacement<,index>



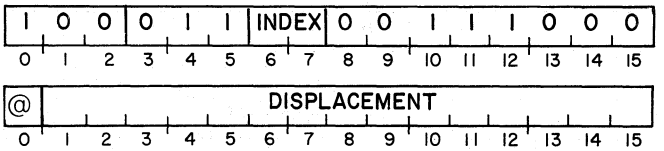
The effective address, "E" is computed and placed in the program counter. Sequential operation continues with the word addressed by the updated value of the program counter.

JUMP TO SUBROUTINE

JSR <@>displacement<,index>



EJSR <@>displacement<,index>

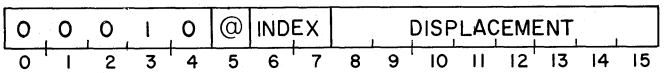


The effective address, "E" is computed. Then the present value of the program counter is incremented by one for JSR and by two for EJSR and the result is placed in AC3. "E" is then placed in the program counter and sequential operation continues with the word addressed by the updated value of the program counter.

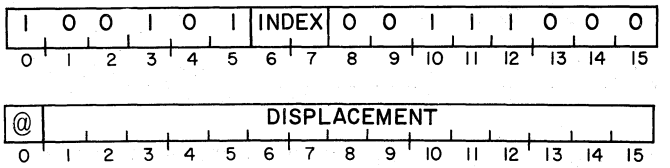
NOTE The computation of "E" is completed before the incremented program counter is placed in AC3.

INCREMENT AND SKIP IF ZERO

ISZ <@>displacement<,index>



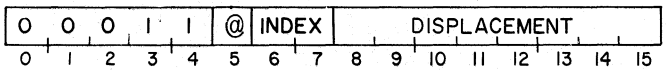
EISZ <@>displacement<,index>



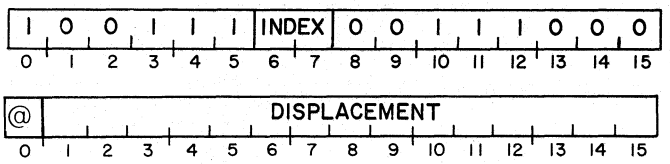
The word addressed by "E" is incremented by one and the result is written back into that location. If the updated value of the location is zero, the next sequential word is skipped.

DECREMENT AND SKIP IF ZERO

DSZ <@>displacement<,index>



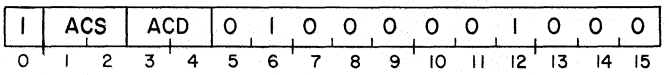
EDSZ <@>displacement<,index>



The word addressed by "E" is decremented by one and the result is written back into that location. If the updated value of the location is zero, the next sequential word is skipped.

SKIP IF ACS GREATER THAN ACD

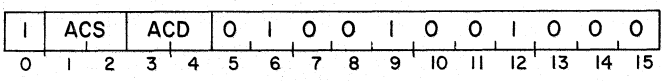
SGT acs,acd



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 IF ACS GREATER THAN OR EQUAL TO ACD

SGE acs,acd

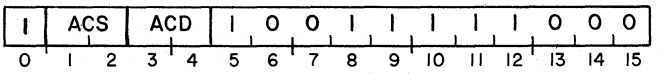


The signed two's complement numbers in ACS and ACD are algebraically compared. If the number in ACS is greater than or equal to the number in ACD, the next sequential word is skipped. The contents of ACS and ACD remain unchanged.

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. For comparison of unsigned integers, the SUBTRACT and ADD COMPLEMENT instructions may be used. Use of these instructions for comparison is described in Appendix H.

COMPARE TO LIMITS

CLM acs,acd



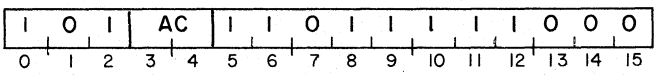
The signed, two's complement number contained in ACS is compared to two signed, two's complement limit values, "L" and "H". If the number in ACS is greater than or equal to L and less than or equal to H, the next sequential word is skipped. If the number in ACS is less than L or greater than H, the next 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 number to be compared is contained in that AC and the limit values L and H are contained 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.

EXECUTE

XCT ac

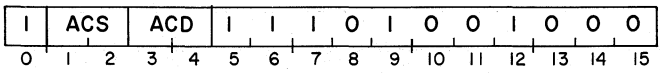


The instruction contained in AC is executed as if it were in main memory in the location occupied by the EXECUTE instruction. If the instruction in AC is an EXECUTE instruction which EXECUTE's the instruction AC, the processor is placed in a one-instruction loop. The Stop switch on the console will not stop the processor, but the Reset switch will.

Due to the possibility of AC containing an EXECUTE instruction, this instruction is interruptable. 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 the programmer a "wait for I/O interrupt" instruction.

SYSTEM CALL

SYC acs,acd



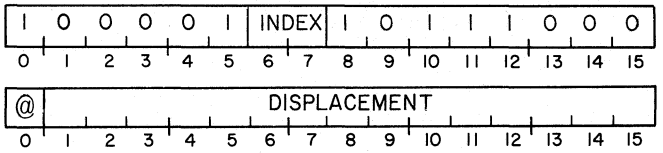
If a user map is enabled, it is disabled and a return block is pushed onto the stack. The program counter in the return block points to the instruction immediately following the SYSTEM CALL instruction. After the return block has been pushed a "jump indirect" to location 2 is executed. If this instruction disabled a user map, then I/O interrupts cannot occur between the time the SYSTEM CALL instruction is executed and the time the instruction pointed to by the contents of location 2 is executed.

NOTE If both accumulators are specified as AC0, no return block is pushed on the stack. The contents of AC0 remain unchanged. If not both accumulators are specified as AC0, then no special action is taken. The contents of the specified accumulators 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.

PUSH JUMP

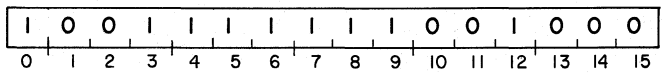
PSHJ <@>displacement<,index>



The address of the next sequential instruction is pushed onto the stack. The effective address "E" is computed and placed in the program counter. Sequential operation continues with the word addressed by the updated value of the program counter.

POP PC AND JUMP

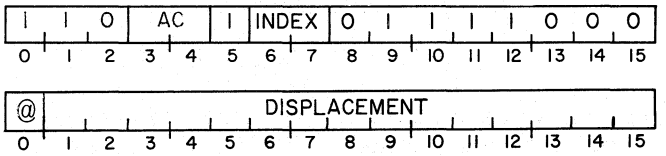
POPJ



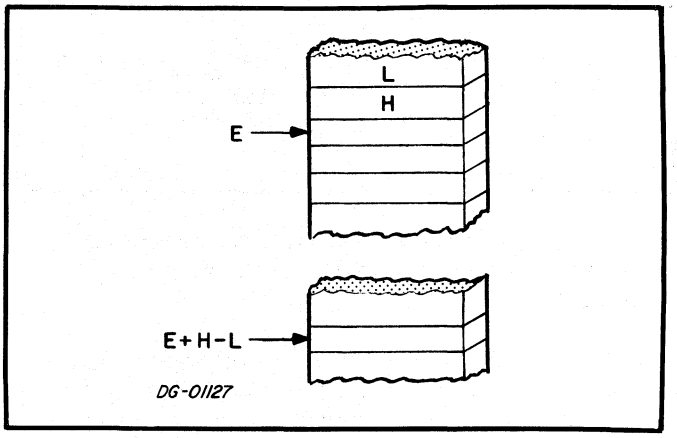
The top word on the stack is popped and placed in the program counter. Sequential operation continues with the word addressed by the updated value of the program counter.

DISPATCH

DSPA ac,<@>displacement<,index>



The effective address "E" is computed. 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.

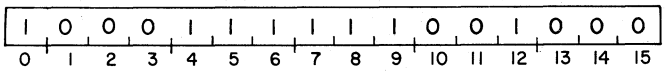


The signed, two's complement number contained in AC is compared to the limit words. If the number in AC is less than L or greater than H, sequential operation continues with the instruction immediately after the DISPATCH instruction.

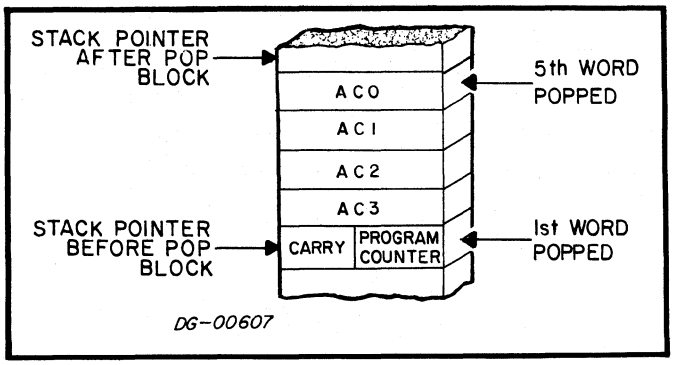
If the number in AC is greater than or equal to L and less than or equal to H, the word at location E-L+number is fetched. If the fetched word is equal to 177777_8, sequential operation continues with the instruction immediately after the DISPATCH instruction. If the fetched word is not equal to 177777_8, this word is treated as the intermediate address in the effective address calculation. After the indirection chain, if any, has been followed, the effective address is placed in the program counter and sequential operation continues with the word addressed by the updated value of the program counter.

POP BLOCK

POPB



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

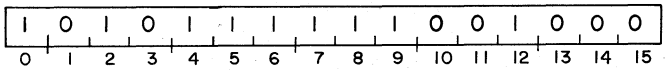


Sequential operation continues with the word addressed by the updated value of the program counter.

The POP BLOCK instruction can be used to return control from routines called by the SUPERVISOR CALL instruction or to return control from an I/O interrupt handler that does not use the stack change facility of the VECTOR ON INTERRUPTING DEVICE CODE instruction.

RETURN

RTN

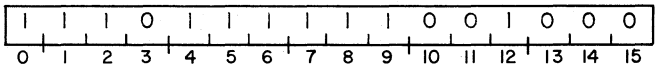


The contents of the frame pointer are placed in the stack pointer and a POP BLOCK instruction is executed. The popped value of AC3 is placed in the frame pointer.

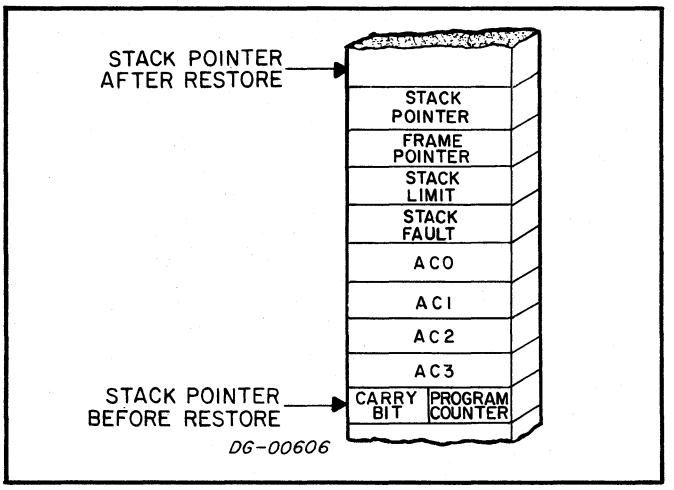
The RETURN instruction can be used to return control from routines that issue a SAVE instruction at their entry points.

RESTORE

RSTR



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



This instruction can be used to return control from an I/O interrupt handler that uses the stack change facility of the VECTOR ON INTERRUPTING DEVICE CODE instruction.

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

NOTE No check for stack underflow is performed as part of the RESTORE operation.

SUBROUTINES CALLS AND RETURNS

The transfer of control between routines is made easier and more orderly by using the stack facility. There are three general ways to effect calls and returns, but more complex ways may be derived. The three basic methods of call and return are discussed here.

The first method transfers control to the subroutine via a JUMP TO SUBROUTINE instruction. The subroutine executes a SAVE instruction at the subroutine entry point and returns control via the RETURN instruction.

```
          ;CALLING PROGRAM
CALL:     JSR   SBRT
          ...
          ...
          ...
          ;SUBROUTINE
SBRT:     SAVE  i
          ...
          ...
          ...
RETURN:   RTN
```

This method has the following characteristics:

1. AC3 of the calling program is destroyed by JSR.
2. The call is only one word.
3. Upon return to the calling program, AC3 contains the calling program's frame pointer.
4. A SAVE instruction is required at each entry point.
5. Arguments are easily passed on the stack, because SAVE sets up the frame pointer for the called routine, and RETURN places the frame pointer of the calling routine in AC3.

The second method transfers control to the subroutine via a JUMP TO SUBROUTINE instruction. The subroutine executes a PUSH MULTIPLE ACCUMULATORS instruction to save the return address and returns control via the POP PC AND JUMP instruction.

```
          ;CALLING PROGRAM
          JSR      SUBR
          ...
          ...
          ...
          ;SUBROUTINE
SUBR:     PSH 3,3
          ...
          ...
RETURN:   POPJ
```

This method has the following characteristics:

1. AC3 of the calling program is destroyed by the JSR.
2. The call is only one word.
3. A PSH 3,3 instruction is required at each entry point.
4. Arguments may be placed in-line in the calling program and conveniently referenced because AC3 points to the first word after the call.

The third method transfers control to the subroutine via a PUSH JUMP instruction. The subroutine returns via a POP PC AND JUMP instruction.

```
          ;CALLING PROGRAM
CALL:     PSHJ SBRT ;PUSH RETURN ADDRESS
                    ; AND JUMP
          ...
          ...
          ;SUBROUTINE
SBRT:     ...
          ...
          ...
RETURN:   POPJ
```

This method has the following characteristics:

1. No accumulators are destroyed.
2. The call requires two words.
3. Multiple entry points are easy to use because no action is required at the entry point.
4. Arguments may be passed in the accumulators.

Example:

An important feature of subroutines that use the stack for saving return information is that they can call themselves without complicated storage allocation procedures. Routines that call themselves are called "recursive" procedures. A good example of a recursive procedure is the factorial function. The number n! (read "n factorial") is equal to 1 if n is equal to either 0 or 1. For values of n greater than 1 the definition of n! is as follows: n! = n*(n-1)*(n-2)*...*1. The function can also be defined as n! = n*((n-1)!). This function can be computed by counting through a loop (iteratively) or by a procedure calling itself (recursively). The following procedure implements the factorial function iteratively. AC2 contains n and AC3 contains the return address. The answer is returned in AC0 and AC1.

```
ENTRY: SUBO    0,0       ;CLEAR AC0
       INC     0,1       ;PUT 1 IN AC1
       ADCZ#   1,2,SNC   ;N GREATER THAN 1?
       JMP     0,3       ;NO--RETURN WITH
                         ; ANSWER = 1
       MUL               ;YES--MULTIPLY
       SBI     1,2       ;DECREASE N BY 1
       MOVZR#  2,2,SNR   ;N=1?
       JMP     0,3       ;YES--RETURN
       JMP     .-4       ;NO--DO IT AGAIN
```

The following procedure implements the factorial function recursively. AC2 contains n. The answer is returned in AC0 and AC1. The procedure is called with a PUSH JUMP.

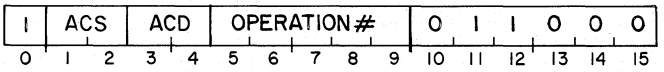
```
ENTRY: SUBO    0,0       ;CLEAR AC0
       INC     0,1       ;PUT 1 IN AC1
       ADCZ#   1,2,SNC   ;N GREATER THAN 1?
       POPJ              ;NO--RETURN
       PSH     2,2       ;YES--SAVE N
       SBI     1,2       ;DECREASE N BY 1
       PSHJ    ENTRY     ;PUSH RETURN AD-
                         ; DRESS AND DO
                         ; FACTORIAL OF N-1
       POP     2,2       ;RETRIEVE N
       MUL               ;MULTIPLY (N-1)! BY
                         ; N
       POPJ              ;RETURN
```

EXTENDED OPERATION FEATURE

The extended operation feature (XOP) provides a general, efficient method of transferring control to and from procedures. When used with the writeable control store feature (WCS) the XOP feature enables the user to transfer control to any one of 16 entry points in WCS. This gives the user the capability of implementing his own specialized instructions and executing them conveniently.

EXTENDED OPERATION

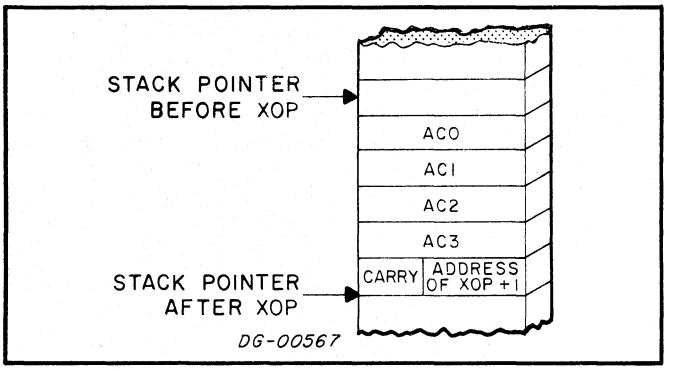
XOP acs,acd,operation number



A return block is pushed onto the stack. The address in the stack of ACS is placed into AC2 and the address in the stack of ACD is placed into AC3. Memory location 44_8 (the XOP origin) must contain the starting address of a 32_{10} word table of addresses. These addresses are the starting location of the various XOP operations.

The operation number in the XOP instruction is added to the contents of the XOP origin to produce the address of a word in the XOP table. That word is fetched and treated as the intermediate address in the effective address calculation. After the indirection chain, if any has been followed, the effective address is placed in the program counter. The contents of AC0, AC1 and the XOP origin remain unchanged.

The format of the return block pushed by the XOP instruction is as follows:



This return block is configured so that the XOP procedure can return control to the calling program via the POP BLOCK instruction.

WRITEABLE CONTROL STORE FEATURE

The writeable control store (WCS) feature of the ECLIPSE line of computers allows the user to transfer control to any one of 16 entry points in WCS. These routines in WCS allow the microprogrammer to utilize the full power of the ECLIPSE line microcode processor.

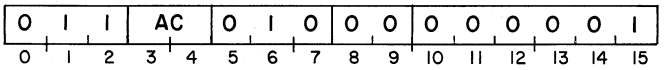
Placing Microcode in WCS

Before the user can utilize the XOP feature to execute instructions in WCS, the microcode must be placed in the WCS locations and the entry points must be specified. This discussion treats only how to place microcode in WCS and how to specify the decode1 and decode2 addresses. For a detailed discussion of how to write microprograms see "Microprogramming With The ECLIPSE computer WCS Feature" (DGC 014-000045).

The setting-up of WCS is done with three I/O instructions. For a detailed discussion of the I/O format instructions see the I/O section of this manual.

SPECIFY ADDRESS

DOA ac, WCS



The contents of the specified AC are transferred to the WCS word register. The format of the information in the specified AC is dependent upon whether the user is transferring microcode words or decode addresses to the WCS feature. If this SPECIFY ADDRESS instruction is to be followed by a LOAD MICROCODE instruction, the format of the specified AC is as follows:

BIT NUMBER	CONTENTS
0-5	Unused
6-13	Address in WCS of the 56-bit microword that will be loaded by the following LOAD MICROCODE instruction.
14-15	Field of the 56-bit microword that will be loaded by the following LOAD MICROCODE instruction. If these bits are 00, the field is microcode bits 0-15. If these bits are 01, the field is microcode bits 16-31. If these bits are 10, the field is microcode bits 32-47. If these bits are 11, the field is microcode bits 48-55.

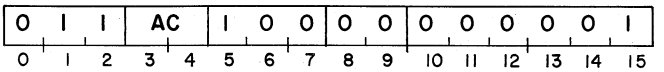
If this SPECIFY ADDRESS instruction is to be followed by a LOAD DECODE ADDRESS instruction, the format of the specified AC is as follows:

BIT NUMBER	CONTENTS
0-10	Unused
11-14	Entry number--from bits 6-9 of of the corresponding XOP1 instruction.
15	Decode number. If this bit is 0, the following LOAD DECODE ADDRESS instruction will specify a decode1 address. If this bit is 1, the following LOAD DECODE ADDRESS instruction will specify a decode2 address.

The contents of the specified AC remain unchanged.

LOAD MICROCODE

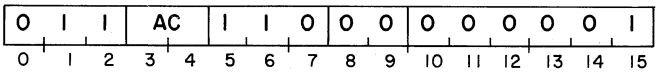
DOB ac, WCS



The contents of the specified AC are placed in the field of the microcode word whose address was specified in the previous SPECIFY ADDRESS instruction. If the field specified in the previous SPECIFY ADDRESS instruction was field 3 (bits 14-15=11), only bits 0-7 of the specified AC are transferred to the WCS feature. The contents of the specified AC remain unchanged.

LOAD DECODE ADDRESS

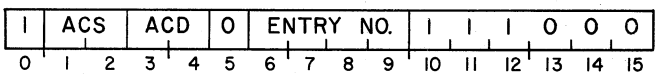
DOC ac, WCS



Bits 8-15 of the specified AC are placed in the decode word whose address was specified in the previous SPECIFY ADDRESS instruction. The contents of the specified AC remain unchanged.

ENTER WCS

XOP1 acs, acd, entry number



The microprogram in WCS whose entry number corresponds to bits 6-9 in the ENTER WCS instruction is executed. The use of the accumulators, whether or not they are changed, and the location of the next instruction are all dependent upon the executed microprogram.

If the WCS feature is not installed, the ENTER WCS instruction operates exactly like the EXTENDED OPERATION instruction except that before the entry number is added to the XOP origin, 32_{10} is added to the entry number.

MEMORY ALLOCATION AND PROTECTION

NOTE *In the following section, "MAP" refers to the Memory Allocation and Protection unit, whereas "map" refers to a set of memory translation functions used by the MAP.*

The MAP unit provides the hardware necessary to control and use more than 32K of physical memory. A MAP is useful for systems with two users and required for systems with more than two users. In addition, the MAP provides protection functions which help protect the integrity of a large system.

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

For the following discussion, certain words and phrases should be defined:

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 MAP to address the physical memory. The physical address space has a maximum size of 131,072 words (128K) and is addressed by a 17-bit address.

Address Translation - The process of translating logical addresses into physical addresses, and vice versa.

Memory Space - The addresses (physical or logical) assigned to a particular user.

Page - 1024_{10} (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 device used by a particular user or device.

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

Address Translation

The primary function of the MAP 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 MAP 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.

Types of Maps

Two types of maps are provided. *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 MAP can hold two user maps, but only one can be enabled at any one time. Thus if there are two users, the user map for each is specified and loaded into the MAP. 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 MAP 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 references is made by the I/O controller making the reference. Those controllers not equipped to make this distinction use data channel map A by default. See the *Programmer's Reference Manual - Peripherals (DGC No. 015-000021).*

Supervisor Mode

So far we have assumed operation in the user mode. The MAP can also operate in the supervisor mode. The supervisor mode is used to analyze protection faults (see below), load new maps, and, in general, perform various MAP control functions. In the supervisor mode, addresses in the range $0\text{-}75777_8$ (which form logical pages 0-30) are not translated. This means that the supervisor program can be as large as 31K and will reside in the lowest section of memory. In the supervisor mode, addresses in the range $76000\text{-}77777_8$ are translated by the special map for supervisor's logical page 31. This allows the supervisor to access portions of user space while in supervisor mode.

MAP Protection Capabilities

In addition to its address translation functions, the MAP 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 are discussed separately below.

Validity 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

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.

A block of addresses is write protected when the map is specified. Write protection can be enabled or disabled at any time by the supervisor.

Indirect Protection

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 proceed with any further instructions, thus effectively halting the system.

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

I/O Protection

I/O protection protects the I/O devices in the system from unauthorized access. In many systems, all I/O operations are performed through operating system calls. 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 at any time.

MAP Protection Faults

When a user attempts to violate one of the enabled types of protection, a protection fault occurs, as follows:

- The current user map is disabled.
- A 5-word return block is pushed onto the system 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, the fifth word will always be the logical address of the instruction following the instruction that caused the fault. For all other types of faults, the fifth word will be a meaningless number.

Load Effective Address Mode

The *Load Effective Address* instruction uses the same instruction codes as some of the I/O instructions. Without some other indication, the LPU would have no way of knowing which instruction was intended. The MAP therefore has a *Lef* mode bit, which switches the mode of the MAP from *Lef* mode to I/O mode. When the *Lef* mode bit is 1, 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, 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. Thus, *Lef* mode can be used for I/O protection. Note, however, that no indication is given if an I/O instruction is interpreted as an *Lef* instruction. The contents of the indicated accumulator will depend on the I/O instruction, but in general, the results will be undesirable.

When 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. The results will depend on the instruction, but will probably be undesirable.

Initial Conditions

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

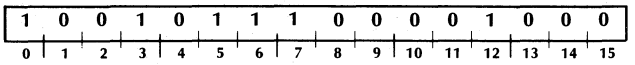
After an *I/O Reset,* the MAP is in supervisor mode, the data channel maps are disabled, and supervisor logical page 31 is mapped to physical page 31.

S/200 AND C/300 MAP INSTRUCTIONS

Listed below are the instructions for the MAP feature found on the ECLIPSE S/200 and C/300 computers.

Load Map

LMP

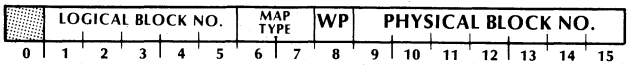


A BLOCK ADD AND MOVE instruction is performed with the exception that no data is written into the destination area. After the contents of AC0 have been added to the fetched word, the result is loaded into the MAP feature. The accumulators are set up in the same manner as for a BLOCK ADD AND MOVE instruction. AC3 is ignored and its contents remain unchanged. If this instruction is issued while in the user mode, it is not executed if I/O protection is enabled. Program operation continues with the next sequential instruction.

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

The information to be loaded into the MAP feature is in three formats. Format number one defines the map for a single 2K byte block of logical memory. Format two defines the I/O devices that are inaccessible to a user. Format three defines the protection features that are to be enabled for a user.

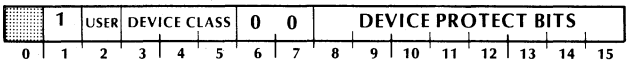
Format Number One - Address Translation



BIT NUMBER	CONTENTS
0	Reserved for future use.
1-5	Logical block number - this is the number of the logical block to be mapped.
6-7	Map type - if 01, this is a translation for the data channel; if 10, this is a translation for user A; if 11, this is a translation for user B.
8	Write Protect - if 1, this block may not be modified if write protection is enabled.
9-15	Physical block number - this is the number of the physical block that corresponds to the logical block given in bits 1-5.

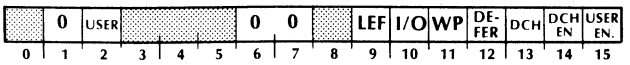
NOTE *A logical block is validity protected by mapping to physical block 127_{10} and setting the write protect bit.*

Format Number Two - I/O Protection



BIT NUMBER	CONTENTS
0	Reserved for future use.
1	Must be 1.
2	User number - if 0, these devices are to be protected from user A; if 1, these devices are to be protected from user B.
3-5	Device class - an unsigned number in the range 0-7 - this is the high-order digit of the two-digit, octal device code
6-7	Format type - must be 00.
8-15	Device protect bits : the second digit of the two-digit device code is specified by the position of the bit in this field. A 1 in any bit protects the corresponding device from receiving any commands directly from the user. For example, if bits 3-5 are 010 and bits 8-15 are 01010000, then devices 21_8 and 23_8 are protected

Format Number Three - Status

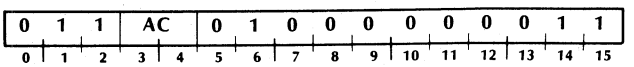


BIT NUMBER	CONTENTS
0	Reserved for future use.
1	Must be 0.
2	User Number - if 0, these status bits refer to user A; if 1, these status bits refer to user B.
3-5	Reserved for future use.
6-7	Format Type - must be 00.
8	Reserved for future use.
9	LEF Mode - if 1, the LOAD EFFECTIVE ADDRESS instruction is to be enabled for this user.
10	I/O Protect - if 1, I/O protection is to be enabled for this user.
11	Write Protect - if 1, write protection is to be enabled for this user.
12	Indirect Protect - if 1, indirect protection is to be enabled for this user.
13	Data Channel Protect - if 1, data channel protection is to be enabled for this user.
14	Data Channel Map Enable - if 1, the data channel map is enabled immediately.
15	User Map Enable - if 1, the user map for this user is enabled after the LOAD MAP instruction is finished.

It is format three that directs the MAP feature to begin translating addresses. If at any time during the execution of the LOAD MAP instruction, the MAP feature receives a word in this format with bit 15 set to 1, the interrupt system is immediately disabled and the map for the user indicated by bit 2 is readied. After the next POP BLOCK, POP PC AND JUMP, RETURN, or STORE instruction or an indirect reference while computing an effective address, the map for the user indicated by bit 2 is enabled. After the first user instruction has started to execute, the interrupt system is enabled. The MAP feature will continue to translate addresses and check for protection violations until directed to stop by a SYSTEM CALL instruction or until it senses a protection violation or an I/O interrupt occurs.

Load Single Word

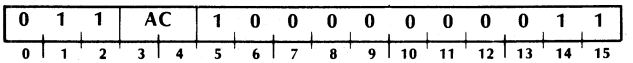
DOA *ac*, MAP



The contents of the specified AC are transferred to the MAP feature. The contents of the specified AC must be in one of the formats listed under the LOAD MAP instruction. The contents of the specified AC remain unchanged.

Map Supervisor Block 31

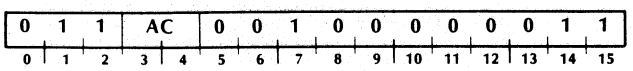
DOB *ac*, MAP



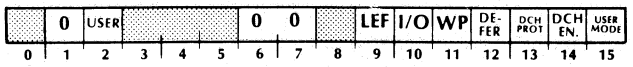
Bits 9-15 of the specified AC are transferred to the MAP feature. These bits specify a physical block number to which logical block 31 will be mapped when in the supervisor mode.

Read User Status

DIA *ac*, MAP



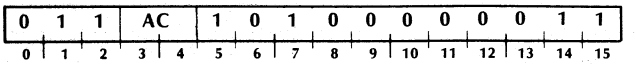
The status of the last enabled user map is placed in the specified AC. The previous contents of the specified AC are lost. The information placed in the specified AC has the following format:



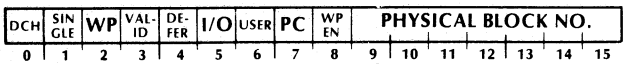
BIT NUMBER	CONTENTS
0	Reserved for future use.
1	Always 0.
2	User number - if 0, these status bits refer to user A; if 1, these status bits refer to user B.
3-5	Reserved for future use.
6-7	Always 0.
8	Reserved for future use.
9	LEF mode - if 1, the LOAD EFFECTIVE ADDRESS is enabled for this user.
10	I/O Protect - if 1, I/O protection is enabled for this user.
11	Write Protect - if 1, write protection is enabled for this user.
12	Indirect Protection - if 1, indirect protection is enabled for this user.
13	Data Channel Protect - if 1, data channel protection is enabled for this user.
14	Data channel Map Enable - if 1, the data channel is currently enabled.
15	User Mode Interrupt - if 1, the last I/O interrupt occurred while in user mode.

Read Map Status

DIC *ac*, MAP



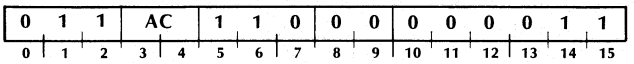
The contents of the MAP status register are placed in the specified AC. The previous contents of the specified AC are lost. The format of the information placed in the specified AC is as follows:



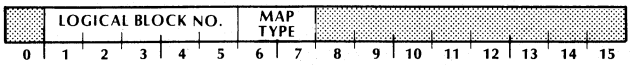
BIT NUMBER	CONTENTS
0	If 1, a data channel protection error has occurred.
1	If 1, the error ocurred during a MAP SINGLE CYCLE instruction.
2	If 1, a write protection error has ocurred for the user indicated in bit 6.
3	If 1, a validity protection error has occurred for the user indicated in bit 6.
4	If 1, an indirect protection error has occurred for the user indicated in bit 6.
5	If 1, an I/O protection error has occurred for the user indicated in bit 6.
6	If 0, the last user map enabled was for user A; if 1, the last user map enabled was for user B.
7	If 1, the program counter pushed onto the system stack is undefined.
8	If 1, write protection is enabled for the physical block whose number is given in bits 9-15.
9-15	This is the physical block number corresponding to the logical page number given in the last TRANSLATE BLOCK instruction.

Translate Block

DOC *ac,* MAP



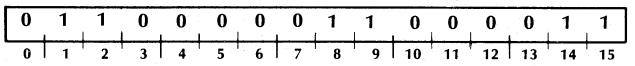
The logical block number in bits 1-5 of the specified AC will be translated to the corresponding physical block number and placed in bits 9-15 of the MAP status register. The contents of the specified AC remain unchanged. The format of the specified AC is as follows:



BIT NUMBER	CONTENTS
0	Reserved for future use.
1-5	Logical block number to be translated.
6-7	If 00, no translation will be performed; If 01, translation will be performed with the map for the data channel. If 10, translation will be performed with the map for user A If 11, translation will be performed with the map for user B.
8-15	Reserved for future use.

Map Single Cycle

NIOP MAP



The last user map enabled is enabled for one memory reference. The first memory reference of the next LOAD or STORE instruction is mapped. After the memory cycle is mapped, the user map is again disabled.

Example:

If AC2 contains 405_8, and the following instruction sequence is issued:

```
        .
        .
NIOP    MAP     ;MAP SINGLE CYCLE
LDA     3,2,2
        .
        .
```

The logical address 407_8 will be mapped using the user map for the last enabled user. The word contained in the corresponding physical location will be placed in AC3.

However, if the following instruction sequence is issued:

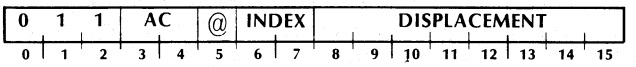
```
        .
        .
NIOP    MAP     ;MAP SINGLE CYCLE
LDA     3,@2,2
        .
        .
```

The logical address 407_8 will be mapped using the user map for the last enabled user. The contents of the corresponding physical location will be used as the first level of an indirection chain. The next memory cycle, which is the second level of the indirection chain, will not be mapped.

NOTE *The interrupt system is disabled from the beginning of the MAP SINGLE CYCLE instruction until after the next LOAD or STORE instruction.*

Load Effective Address

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



If the LEF MODE bit in the user status is 1 for a user, then all I/O instructions issued by that user will be interpreted as LOAD EFFECTIVE ADDRESS instructions.

When a LOAD EFFECTIVE ADDRESS instruction is issued, the logical effective address is computed from bits 5-15 of the instruction and placed in the specified AC. The previous contents of the specified AC are lost. If an auto-incrementing or auto-decrementing location is referenced in the course of the effective address calculation, it is incremented or decremented.

Examples:

INSTRUCTION	RESULT
LEF 0,TABLE	The logical address of TABLE is placed in AC1
LEF 2,34,2	34_8 is added to the unsigned integer in AC2.
LEF 1,-55, 3	55_8 is subtracted from the unsigned integer in AC3 and the result is placed in AC1.
LEF 0,.+0	The logical address of this LOAD EFFECTIVE ADDRESS instruction is placed in AC0.

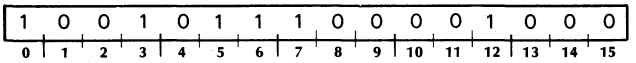
NOTE *The LOAD EFFECTIVE ADDRESS instruction can only be issued in the user mode.*

S/230 AND C/330 MAP INSTRUCTIONS

Listed below are the instructions for the MAP feature found on the ECLIPSE S/230 and C/330 computers.

Load Map

LMP



Successive words from memory are loaded into the MAP feature where they are used to define an address translation function. Words are loaded in consecutive, ascending order according to their addresses. The number of words to be loaded and the address of the beginning of the field are contained in accumulators 1 and 2. Which address translation function is being loaded is determined by the contents of the Map field in the MAP status register.

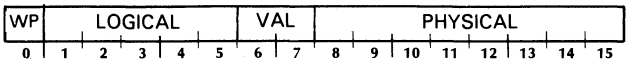
AC0 must contain 0. AC1 must contain an unsigned integer which is the number of words to be loaded into the MAP feature. Bit 0 of AC2 must be set to 0. Bits 1-15 of AC2 must contain the address of the first word to be loaded. AC3 is ignored and its contents remain unchanged.

For each word loaded, the count in AC1 is decremented by one and the source address in AC2 is incremented by 1. Upon completion of the instruction, AC0 and AC1 contain 0 and AC2 contains the address of the word following the last word loaded.

This instruction is interruptable in the same manner as the BLOCK ADD AND MOVE instruction. If this instruction is issued while in the user mode, it is not executed if I/O protection is enabled. Program operation continues with the next sequential instruction.

The words loaded into the MAP feature define the address translation functions for the various user and data channel maps. Which map is to be affected by a LOAD MAP instruction is determined by the contents of the Map field in the MAP status register. This field can be altered by both the LOAD MAP STATUS and the INITIATE PAGE CHECK instruction.

The format of the words loaded into the MAP feature is as follows:

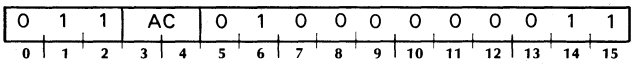


Bits	Name	Function
0	Write Protect	If 1, this logical block will be write protected when write protection is enabled.
1-5	Logical Block	This is the number of the logical block which is to be mapped.
6-7	Validity	These two bits must be be set to 1 when declaring a logical block invalid. Otherwise, they must be set to 0.
8-15	Physical Block	This is the number of the physical block of memory that will hold the logical block defined by bits 1-5.

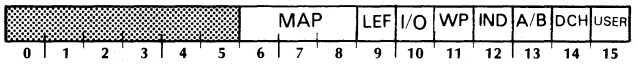
NOTE *A logical block is declared invalid by setting the Write Protect bit to 1 and all of bits 6-15 to 1.*

Load MAP Status

DOA *ac,MAP*



The contents of the specified AC are placed in the MAP status register. The contents of the specified AC remain unchanged. The format of the specified AC is as follows:



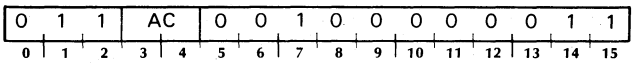
Bits	Name	Function
0-5	----	Reserved for future use.
6-8	Map	Specify which map will be loaded by the next LOAD MAP instruction as follows:
	000	User A
	001	Reserved for future use
	010	User B
	011	Reserved for future use
	100	Data channel A
	101	Data channel C
	110	Data channel B
	111	Data channel D
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, the next user map enabled will be that for user A. If 1, the next user map enabled will be that for user B.
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.

NOTE *If the* Load Map Status *instruction sets the User Enable bit to 1, the interrupt system is inhibited and the MAP waits for an indirect reference or a return type instruction. The interrupt system is released and the MAP begins translating addresses (using the user map specified by bit 13 of the MAP status register):*

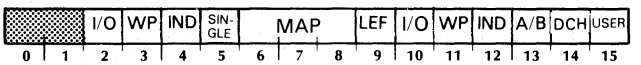
- *After the first level of the next indirect reference: or*
- *Upon completion of a* Pop Block, Pop Jump, Return, *or* Restore *instruction that does not result in a stack fault.*

Read MAP Status

DIA *ac,MAP*



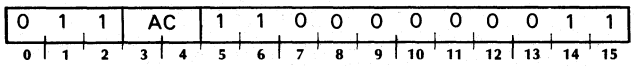
The contents of the MAP status register are placed in the specified AC. The previous contents of the specified AC are lost. The format of the information placed in the specified AC is as follows:



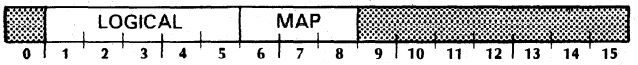
Bits	Name	Contents
0-1	----	Reserved for future use.
2	I/O	If 1, the last protection fault was an I/O protection fault.
3	WP	If 1, the last protection fault was a write protection fault.
4	IND	If 1, the last protection fault was an indirect protection fault.
5	Single Cycle	If 1, the last protection fault occured during a MAP SINGLE CYCLE instruction.
6-8	Map	Specify which map will be loaded by the next LOAD MAP instruction as follows:
	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, the next user map enabled will be that for user A. If 1, the next user map enabled will be that for user B.
14	DCH Enable	If 1, the mapping of data channel addresses will be enabled immediately after this instruction.
15	User Mode	If 1, the last I/O interrupt occurred while in user mode.

Initiate Block Check

DOC *ac,MAP*



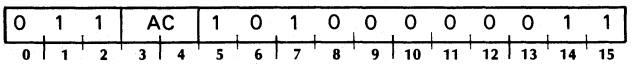
The contents of the specified AC are transferred to the MAP feature for later use by the BLOCK CHECK instruction. The contents of the specified AC remain unchanged. The format of the specified AC is as follows:



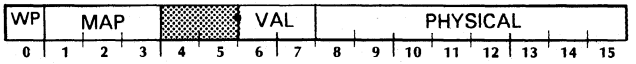
Bits	Name	CONTENTS
0	----	Reserved for future use.
1-5	Logical Block	Number of the logical block for which the check is requested.
6-8	Map	Specify which map should be used for the check as follows: 000 User A 001 Reserved for future use. 010 User B 011 Reserved for future use. 100 Data channel A 101 Data channel C 110 Data channel B 111 Data channel D
9-15	----	Reserved for future use.

Block Check

DIC *ac,MAP*



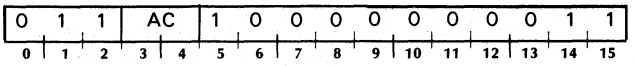
The number of the physical block which corresponds to the logical block specified by the preceeding INITIATE BLOCK CHECK instruction is placed in bits 8-15 of the specified AC. Additional information about the correspondence is placed in bits 0-7. The previous contents of the AC are lost. The format of the information placed in the specified AC is as follows:



Bits	Name	Contents
0	WP	The write protect bit for the logical block which corresponds to the physical block specified by bits 8-15.
1-3	Map	The map which was used to perform the translation between logical block number and physical block number as follows: 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
4-5	----	Reserved for future use
6-7	Validity	If both these bits are 1, the logical block which corresponds to the physical block specified by bits 8-15 is validity protected.
8-15	Physical Block	The number of the physical block which corresponds to the logical block given in the preceeding INITATE BLOCK CHECK instruction.

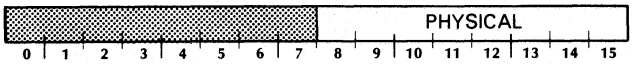
Map Supervisor Block 31

DOB *ac,MAP*



Bits 8-15 of the specified AC are transferred to the MAP feature. These bits specify a physical block number to which logical block 31 will be mapped when in the supervisor mode.

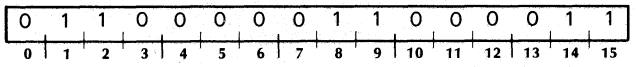
The contents of the specified AC remain unchanged. The format of the specified AC is as follows:



Bits	Name	Contents
0-7	----	Reserved for future use.
8-15	Physical Block	The number of the physical block to which logical block 31 should be mapped when in supervisor mode.

Map Single Cycle

NIOP MAP



The user map is enabled for one memory reference. The first memory reference of the next LDA or STA instruction is mapped. After the memory cycle is mapped, the user map is again disabled.

Example:

If AC2 contains 405_8, and the following instruction sequence is issued:

```
.
.
NIOP    MAP     ;MAP SINGLE CYCLE
LDA     3,2,2
.
.
```

The logical address 407_8 will be mapped using the user map specified by bit B of the MAP status register at the time of the memory reference. The word contained in the corresponding physical location will be placed in AC3.

However, if the following instruction sequence is issued:

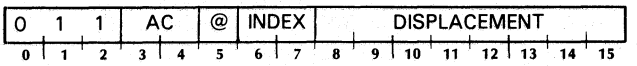
```
.
.
NIOP    MAP     ;MAP SINGLE CYCLE
LDA     3, @ 2,2
.
.
```

The logical address 407_8 will be mapped using the user map for the last enabled user. The contents of the corresponding physical location will be used as the first level of an indirection chain. The next memory cycle, which is the second level of the indirection chain, will not be mapped.

NOTE *The interrupt system is disabled from the beginning of the MAP SINGLE CYCLE instruction until after the next LDA or STA instruction.*

Load Effective Address

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



If the LEF MODE bit in the user status is 1 for a user, then all I/O instructions issued by that user will be interpreted as LOAD EFFECTIVE ADDRESS instructions, regardless of the state of the I/O protection bit in the user status.

When a LOAD EFFECTIVE ADDRESS instruction is issued, the logical effective address is computed from bits 5-15 of the instruction and placed in the specified AC. The previous contents of the specified AC are lost. If an auto-incrementing or auto-decrementing location is referenced in the course of the effective address calculation, it is incremented or decremented.

Examples:

INSTRUCTION	RESULT
LEF 0,TABLE	The logical address of TABLE is placed in AC0.
LEF 2,34,2	34_8 is added to the unsigned integer in AC2
LEF 1,-55, 3	55_8 is subtracted from the unsigned integer in AC 3 and the result is placed in AC1
LEF 0,.+0	The logical address of this LOAD EFFECTIVE ADDRESS instruction is placed in AC0

NOTE *The LOAD EFFECTIVE ADDRESS instruction can only be issued in the user mode.*

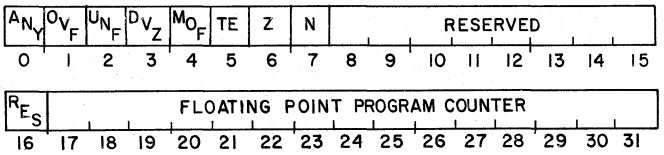
FLOATING POINT ARITHMETIC

In addition to performing fixed point arithmetic, the ECLIPSE line of computers can also perform floating point arithmetic. This feature provides the capability to perform rapid and convenient arithmetic operations on numbers with a much larger range than would be feasible using the fixed point arithmetic instruction set. The precision with which these numbers can be manipulated exceeds the precision readily available with the fixed point instruction set.

If the floating point feature is not installed, instructions in the floating point arithmetic instruction set will be executed as "NO OPS", i.e., "JMP .+1".

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 FAC0, FAC1, FAC2, and FAC3. The FPSR is a 32-bit register that contains information about the present status of the floating point processor. The format of the FPSR is given below.



BIT	MNEMONIC	DESCRIPTION
0	ANY	Indicates that any of bits 1-4 are set.
1	OVF	Overflow indicator meaning that during processing of a floating point instruction, an exponent overflow occurred. The result is correct except that the exponent is 128 too small.
2	UNF	Underflow indicator meaning that during processing of a floating point instruction, an exponent underflow occurred. The result is correct except that the exponent is 128 too large.
3	DVZ	During a floating point divide, a zero divisor was detected. The division was aborted and the operands remain unchanged.
4	MOF	Mantissa overflow indicator meaning that during a floating point instruction, a bit of significance was shifted out of the high order end of the mantissa. Also set if, during a FIX instruction, the result cannot fit into the destination location.
5	TE	Trap enable. If this bit is 1, the setting of 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 equal to zero.
7	N	Negative bit. The result of the last floating point operation was less than zero.
8-16	RES	RESERVED
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.

Floating Point Faults

Upon completion of any floating point instruction, if any of bits 1-4 in the FPSR are set, a floating point fault is indicated. If bit 5 in the FPSR is also set, a floating point trap is initiated. Upon issuance of the next floating point instruction, if it is not a PUSH FLOATING POINT STATE or POP FLOATING POINT STATE instruction, a floating point fault will occur. A return block is pushed onto the stack and a "jump indirect" to location 45_8 instruction is executed. It is assumed that if bit 5 in the FPSR is 1, memory location 45_8 contains the address of the floating point fault handler. The return block pushed in the event of a floating point fault has the following format:

WORD # PUSHED	DESCRIPTION
1	AC0
2	AC1
3	AC2
4	AC3
5	Bit 0 = carry bit Bits 1-15 = return address

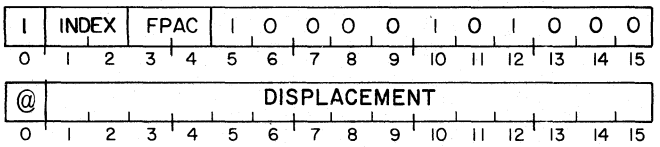
NOTE This is not the address of the Floating Point instruction that caused the fault. It is the address of the next user instruction to be executed.

Because PUSH FLOATING POINT STATE or POP FLOATING POINT STATE save the flags in the FPSR, and because bits 1-4 are tested for possible fault conditions after every floating point operation, a floating point trap always occurs in the environment of the program that caused the fault.

FLOATING POINT INSTRUCTIONS

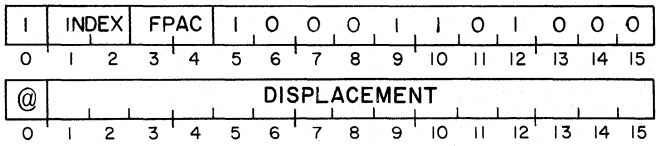
LOAD FLOATING POINT SINGLE

FLDS fpac,<@>displacement<,index>



LOAD FLOATING POINT DOUBLE

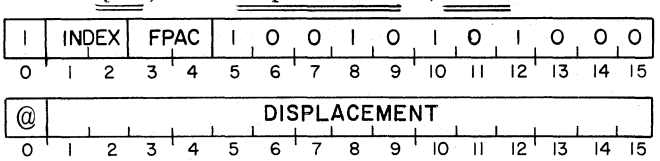
FLDD fpac,<@>displacement<,index>



The effective address "E" is computed. The floating point number at that address is placed in FPAC. For single precision, the low-order 32 bits of FPAC are set to 0. The previous contents of FPAC are lost. The Z and N bits in the FPSR are set to reflect the new contents of FPAC.

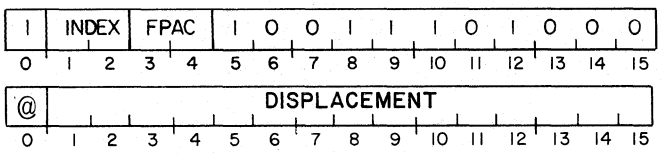
STORE FLOATING POINT SINGLE

FSTS fpac,<@>displacement<,index>



STORE FLOATING POINT DOUBLE

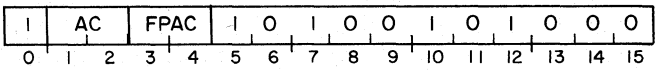
FSTD fpac,<@>displacement<,index>



The effective address "E" is computed and the floating point number contained in FPAC is placed at the memory location addressed by E. For single precision, only the high-order 32 bits of FPAC are stored. The contents of FPAC remain unchanged. The previous contents of the addressed memory location are lost. The condition codes in the FPSR remain unchanged.

FLOAT FROM AC

FLAS ac,fpac

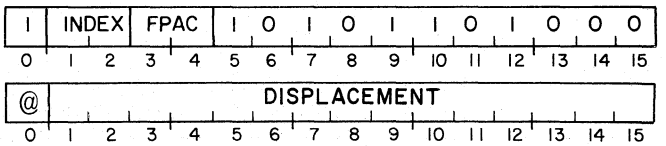


The signed two's complement number contained in AC is converted to a single precision floating point number and placed in FPAC. The low-order 32 bits of FPAC are set to 0. The contents of AC remain unchanged. The previous contents of FPAC are lost. The Z and N bits in the FPSR are set to reflect the new contents of FPAC.

The range of numbers that can be converted is $-32,768_{10}$ to $+32,767_{10}$.

FLOAT FROM MEMORY

FLMD fpac,<@>displacement<,index>

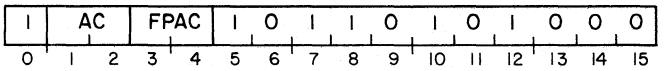


The effective address "E" is computed. The 32-bit, signed, two's complement number addressed by E is converted to a double precision floating point number and placed in FPAC. The previous contents of FPAC are lost. The Z and N bits in the FPSR are set to reflect the new contents of FPAC.

The range of numbers that can be converted is $-2,147,483,648_{10}$ to $+2,147,483,647_{10}$.

FIX TO AC

FFAS ac,fpac

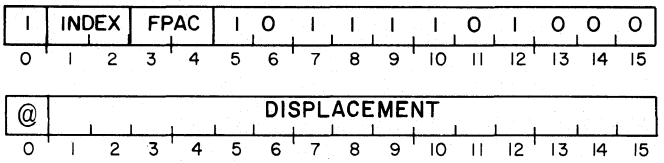


The integer portion of the floating point number contained in FPAC is converted to a signed two's complement number and placed in AC. If the magnitude of the number in FPAC is such that it will not fit into AC, the MOF bit is set in the FPSR and the sign bit and the low-order 15 bits of the converted number are placed in AC. The contents of FPAC remain unchanged. The Z and N bits in the FPSR are both set to 0. The previous contents of AC are lost.

The range of numbers than can be converted without overflow is $-32,767_{10}$ to $+32,767_{10}$.

FIX TO MEMORY

FFMD fpac,<@>displacement<,index>

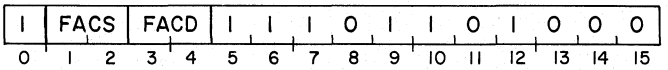


The effective address "E" is computed. The integer portion of the floating point number contained in FPAC is converted to a 32-bit, signed, two's complement number and placed in the memory location addressed by E. If the magnitude of the converted number is such that it will not fit into 32 bits, the MOF bit is set in the FPSR and the sign bit and the low-order 31 bits of the converted number are placed in the location addressed by E. The contents of FPAC remain unchanged. The Z and N bits in the FPSR are set to 0.

The range of numbers that can be converted without overflow is $-2,147,483,647_{10}$ to $+2,147,483,647_{10}$.

MOVE FLOATING POINT

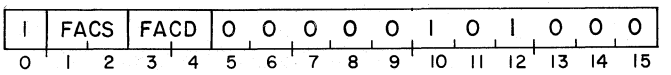
FMOV facs,facd



The contents of FACS are placed in FACD. The previous contents of FACD are lost. The contents of FACS remain unchanged. The Z and N bits in the FPSR are set to reflect the new contents of FACD.

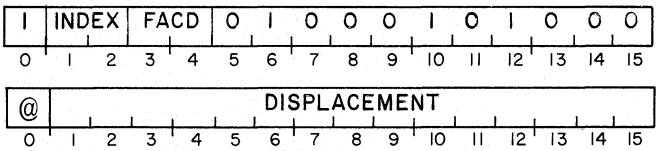
ADD SINGLE (FPAC to FPAC)

FAS facs,facd



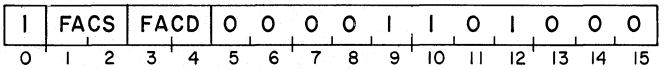
ADD SINGLE (memory to FPAC)

FAMS facd,<@>displacement<,index>



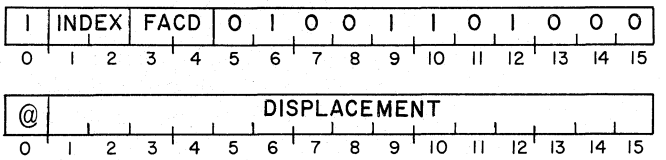
ADD DOUBLE (FPAC to FPAC)

FAD facs,facd



ADD DOUBLE (memory to FPAC)

FAMD facd,<@>displacement<,index>



The floating point number contained in the source location is added to the floating point number in FACD and the normalized result is placed in the FACD. The previous contents of FACD are lost. The contents of the source location remain unchanged. The Z and N bits in the FPSR are set to reflect the new contents of FACD.

For an add from memory, the effective address "E" is computed. E addresses either a 2-word (single precision) or 4-word (double precision) operand. For a single precision add, the operand from memory is extended with 32 low-order zeroes before the operation takes place.

In order to achieve greater accuracy, all 64 bits of FACD take part in a single precision add. If the source operand in a single precision add is contained in an FPAC, then all 64 bits of this number also participate in the add.

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. 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 14 hex digits. If this condition occurs, no normalization takes place.

After alignment, the mantissas are added together. The result of this addition is termed 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 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 FPSR, and the number in FACD is correct except that the exponent is 128 too small.

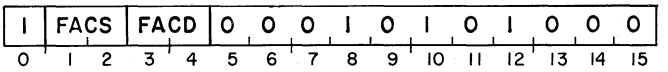
If there is no mantissa overflow, the mantissa of the intermediate result is examined for leading hex zeroes. If the mantissa is found to be all zeroes, a true zero is placed in the FACD and the instruction is terminated.

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 FPSR and the instruction is terminated. The number in the FACD is correct except that the exponent is 128 too large.

For single precision, the low-order 32 bits of the result are set to 0.

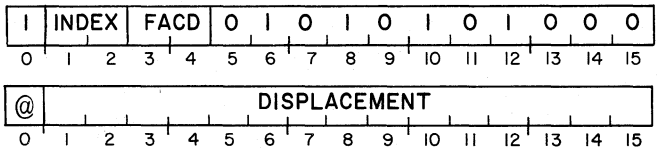
SUBTRACT SINGLE (FPAC from FPAC)

FSS facs, facd



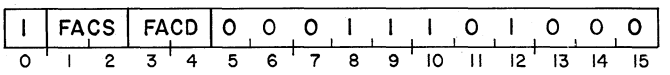
SUBTRACT SINGLE (memory from FPAC)

FSMS facd, <@>displacement<, index>



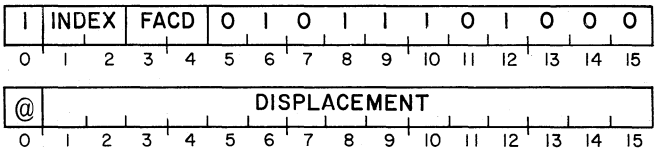
SUBTRACT DOUBLE (FPAC from FPAC)

FSD facs, facd



SUBTRACT DOUBLE (memory from FPAC)

FSMD facd, <@>displacement<, index>



The floating point number contained in the source location is subtracted from the floating point number in FACD and the normalized result is placed in the FACD. The previous contents of FACD are lost. The contents of the source location remain unchanged. The Z and N bits in the FPSR are set to reflect the new contents of FACD.

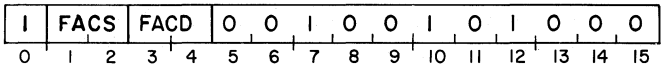
For a subtract from memory, the effective address "E" is computed. E addresses either a 2-word (single precision) or 4-word (double precision) operand. For a single precision subtract, the operand from memory is extended with 32 low-order zeroes before the operation takes place.

In order to achieve greater accuracy, all 64 bits of FACD take part in a single precision subtract. 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.

For single precision, the low-order 32 bits of the result are set to 0.

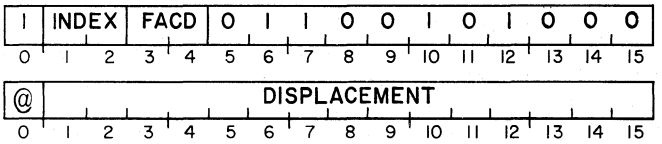
MULTIPLY SINGLE (FPAC by FPAC)

FMS facs, facd



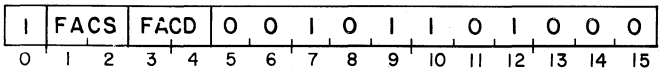
MULTIPLY SINGLE (FPAC by memory)

FMMS facd, <@>displacement<, index>



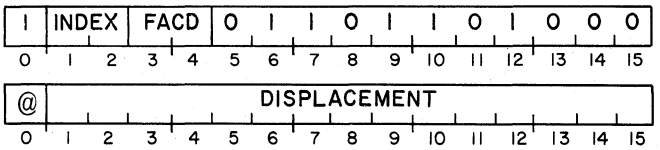
MULTIPLY DOUBLE (FPAC by FPAC)

FMD facs, facd



MULTIPLY DOUBLE (FPAC by memory)

FMMD facd, <@>displacement<, index>



The floating point number contained in FACD is multiplied by the floating point number contained in the source location and the normalized result is placed in FACD. The previous contents of FACD are lost. The contents of the source location remain unchanged. The Z and N bits in the FPSR are set to reflect the new contents of FACD.

For a multiply from memory, the effective address "E" is computed. E addresses either a 2-word (single precision) or 4-word (double precision) operand. For a single precision multiply, the operand from memory is extended with 32 low-order zeroes before the operation takes place.

In order to achieve greater accuracy, all 64 bits of FACD take part in a single precision multiply. If the source operand in a single precision multiply is contained in an FPAC, then only the high-order 32 bits of this number participate in the multiply.

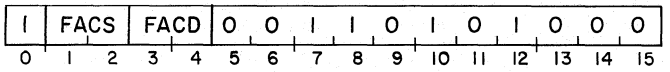
The mantissas of the two numbers are multiplied together to give the mantissa of the intermediate result. The exponents of the two numbers are added together and 64 is subtracted. This subtraction of 64 maintains the "Excess 64" notation. The result of the exponent manipulation becomes the exponent of the intermediate result. The sign of the intermediate result is determined from the sign 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 bit in the FPSR is set. 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.

For single precision, the low-order 32 bits of the result are set to 0.

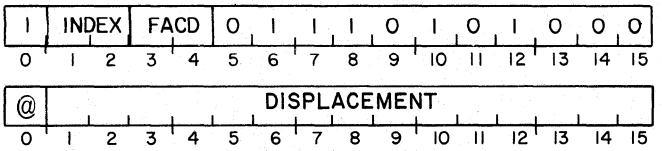
DIVIDE SINGLE (FPAC by FPAC)

FDS facs, facd



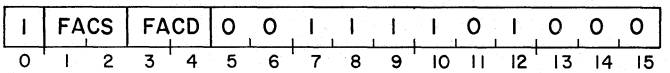
DIVIDE SINGLE (FPAC by memory)

FDMS facd, <@>displacement<, index>



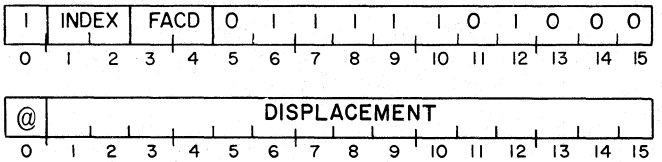
DIVIDE DOUBLE (FPAC by FPAC)

FDD facs, facd



DIVIDE DOUBLE (FPAC by memory)

FDMD facd, <@>displacement<, index>



The floating point number contained in FACD is divided by the floating point number contained in the source location and the result is placed in FACD. The previous contents of FACD are lost. The contents of the source location remain unchanged. The Z and N bits in the FPSR are set to reflect the new contents of FACD. Because the operands are assumed to be normalized, and the division produces a normalized result with normalized operands, no normalization takes place.

For a divide from memory, the effective address "E" is computed. E addresses either a 2-word (single precision) or 4-word (double precision) operand. For a single precision divide, the operand from memory is extended with 32 low-order zeroes before the operation takes place.

In order to achieve greater accuracy, all 64 bits of FACD take part in a single precision divide; however, only 24 quotient bits are formed. For single precision, the low-order 32 bits of the result are set to 0.

The source operand is checked for a zero mantissa. If the mantissa is zero, the DVZ bit is set in the FPSR and the instruction is terminated. The number in FACD remains unchanged. 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. This process continues until the mantissa of the number in FACD is less than the mantissa of the source operand.

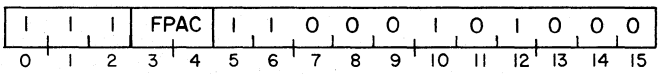
The mantissa in FACD is then divided by the mantissa of the source operand and the quotient is the mantissa of the intermediate result. The exponent of the source operand is subtracted from the exponent in FACD and 64 is added to this result. This addition of 64 maintains the "Excess 64" notation. The 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.

If the exponent processing produces either overflow or underflow, the corresponding bit in the FPSR 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.

For single precision, the low-order 32 bits of the result are set to 0.

NEGATE

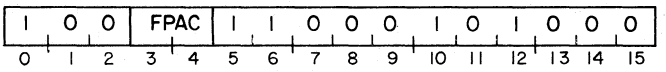
FNEG fpac



The sign bit of FPAC is inverted. Bits 1-63 of FPAC remain unchanged. The Z and N bits in the FPSR are set to reflect the new contents of FPAC. If FPAC contains true zero, the sign bit remains unchanged.

NORMALIZE

FNOM fpac

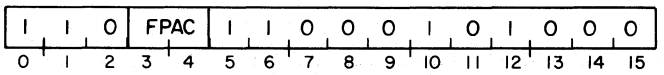


The floating point number in FPAC is normalized. If all bits of the mantissa are zero, a true zero is set in FPAC. If an exponent underflow occurs, the UNF bit is set in the FPSR. The number in FPAC is correct, except that the exponent is 128 too large.

The Z and N bits in the FPSR are set to reflect the new contents of FPAC.

ABSOLUTE VALUE

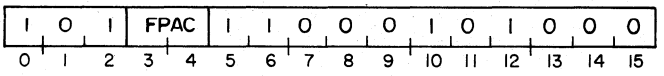
FAB fpac



The sign bit of FPAC is set to 0. Bits 1-63 of FPAC remain unchanged. The Z and N bits in the FPSR are set to reflect the new contents of FPAC.

READ HIGH WORD

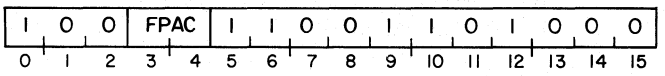
FRH fpac



The high-order 16 bits of FPAC are placed in AC0. The previous contents of AC0 are lost. The contents of FPAC and the Z and N bits in the FPSR remain unchanged.

SCALE

FSCAL fpac



The mantissa of the floating point number in FPAC is shifted either right or left, depending upon the contents of bits 1-7 of AC0. The contents of AC0 remain unchanged.

Bits 1-7 of AC0 are treated as an exponent in "Excess 64" representation. The difference between this exponent and the exponent in FPAC is computed by subtracting the exponent in FPAC from the number contained in AC0 bits 1-7. If the difference is zero, the instruction is terminated. If the difference is positive, the mantissa contained in FPAC is shifted right that number of hex digits. If the difference is negative, the mantissa contained in FPAC is shifted left that number of hex digits and the MOF bit in the FPSR is set. After the shift, the contents of bits 1-7 of AC0 replace the exponent contained in FPAC.

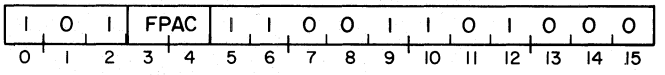
Bits shifted out of either end of the mantissa are lost.

If the entire mantissa is shifted out of FPAC, FPAC is set to true zero.

The Z and N bits in the FPSR are set to reflect the new contents of FPAC.

LOAD EXPONENT

FEXP fpac



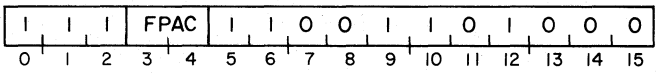
Bits 1-7 of AC0 replace bits 1-7 of FPAC. Bits 0 and 8-15 of AC0 are ignored. Bits 0 and 8-63 of FPAC remain unchanged. The entire contents of AC0 remain unchanged. The Z and N bits in the FPSR are set to reflect the new contents of FPAC.

If FPAC contains true zero, bits 1-7 of FPAC remain unchanged.

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

HALVE

FHLV fpac



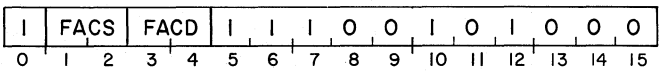
The mantissa contained in FPAC is shifted right one bit position. The vacated bit position is filled with a zero and the bit shifted out is lost. The number is then normalized and the result is placed in FPAC. If the normalization process causes an exponent underflow, the UNF bit in the FPSR is set and the number in FPAC is correct, except that the exponent is 128 too large.

The Z and N bits in the FPSR are set to reflect the new contents of FPAC.

NOTE The effect of this instruction is to divide the floating point number contained in FPAC by 2.

COMPARE FLOATING POINT

FCMP facs, facd

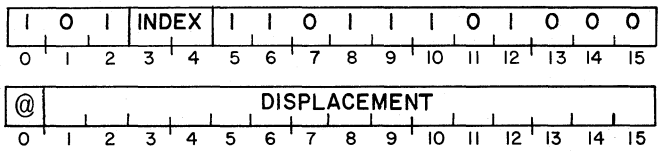


The floating point numbers in FACS and FACD are algebraically compared to each other and the Z and N bits in the FPSR are set to reflect the result. The contents of FACS and FACD remain unchanged. The results of the compare and the corresponding bit settings are as follows:

RESULT	BIT SETTINGS Z	N
FACS = FACD	1	0
FACS > FACD	0	1
FACS < FACD	0	0

LOAD FLOATING POINT STATUS

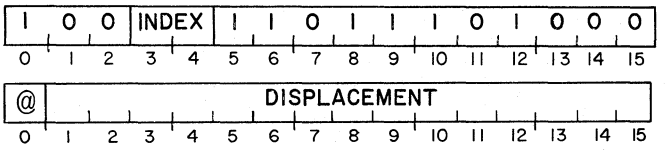
FLST <@>displacement<,index>



The effective address "E" is computed. The 32-bit operand addressed by E is placed in the FPSR. The condition codes are set to the values of the loaded bits.

STORE FLOATING POINT STATUS

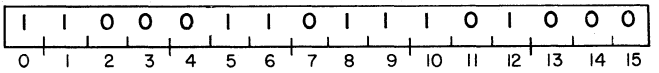
FSST <@>displacement<,index>



The effective address "E" is computed. The 32-bit contents of the FPSR are placed in the memory location addressed by E. The contents of the FPSR remain unchanged.

TRAP ENABLE

FTE

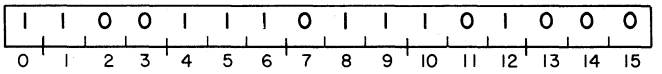


The trap enable bit in the FPSR is set to 1.

NOTE When a FLOATING POINT FAULT occurs and the trap enable bit is 1, it 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.

TRAP DISABLE

FTD

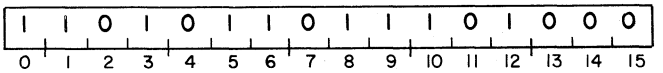


The trap enable bit in the FPSR is set to 0.

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

CLEAR ERRORS

FCLE

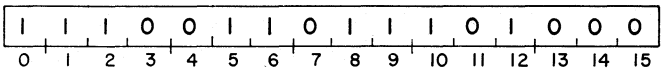


Bits 0-4 of the FPSR are set to 0.

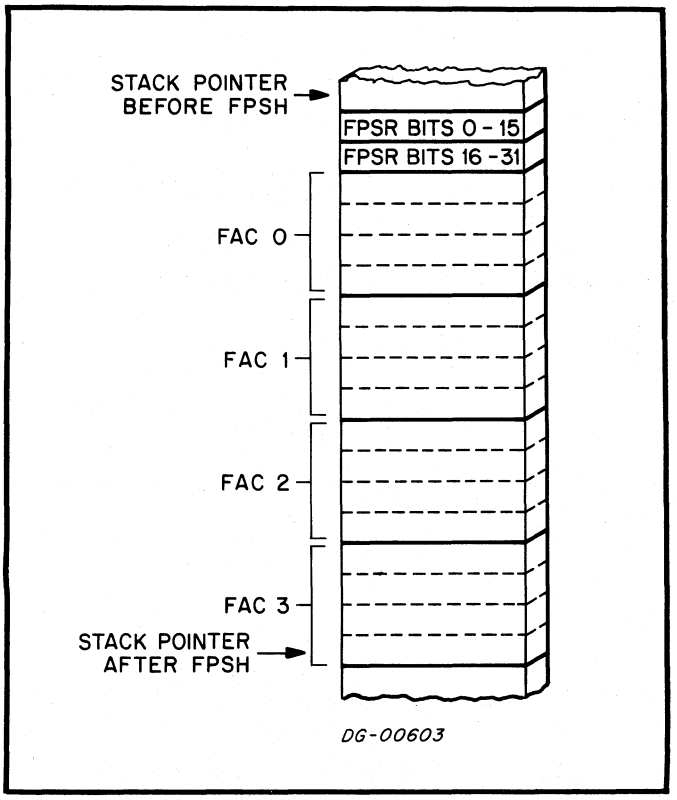
NOTE The I/O RESET instruction will set these bits to 0.

PUSH FLOATING POINT STATE

FPSH



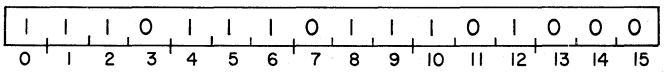
An 18 word floating point state block is pushed onto the user stack. The format of the 18 words pushed is as follows:



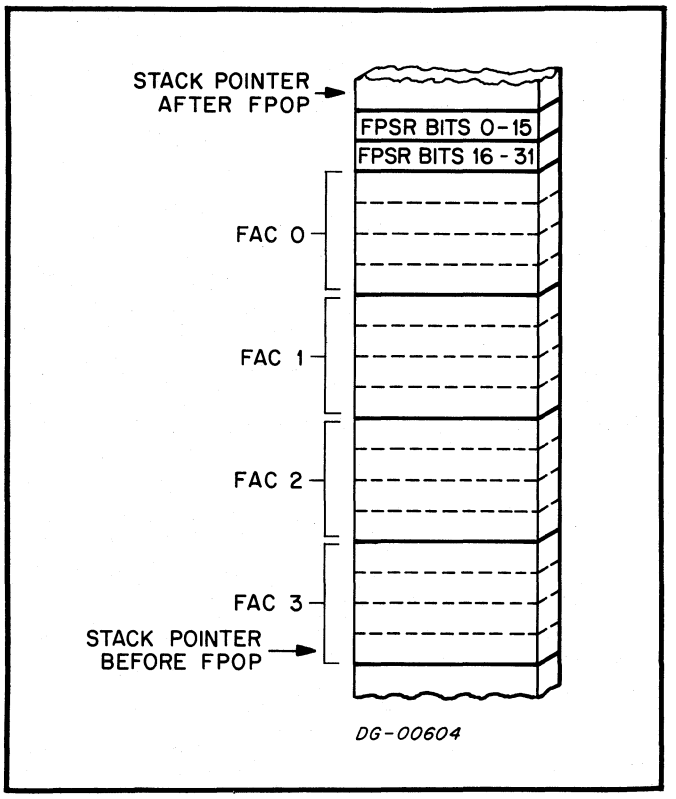
The contents of the floating point accumulators and the FPSR remain unchanged.

POP FLOATING POINT STATE

FPOP



The state of the floating point unit is altered by popping 18 words off of the user stack. The words popped and their destinations are as follows:



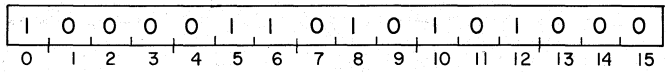
NOTE Due to the potentially long time required to perform a PUSH FLOATING POINT STATE or POP FLOATING POINT STATE, in relation to I/O interrupt requests, these instructions are interruptable. Because the stack pointer and program counter are not updated until the completion of these instructions, any interrupt service routines that return control to the interrupted program via the program counter stored in location 0 will correctly restart these instructions.

Arithmetic Test

There are eight instructions in the floating point instruction set that test the Z and N bits in the FPSR and skip on the result of the test. These instructions are described below.

NO SKIP

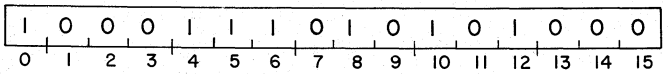
FNS



The next sequential word is executed.

SKIP ALWAYS

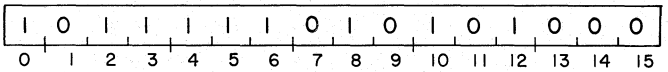
FSA



The next sequential word is skipped.

SKIP ON GREATER THAN ZERO

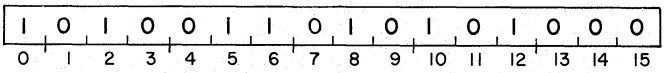
FSGT



If both the Z and N bits in the FPSR are 0, the next sequential word is skipped.

SKIP ON LESS THAN ZERO

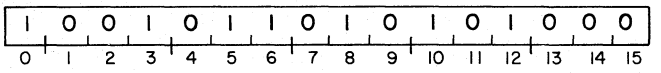
FSLT



If the N bit in the FPSR is 1, the next sequential word is skipped.

SKIP ON ZERO

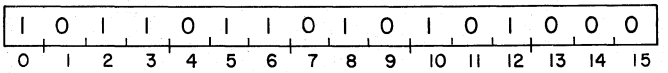
FSEQ



If the Z bit in the FPSR is 1, the next sequential word is skipped.

SKIP ON LESS THAN OR EQUAL TO ZERO

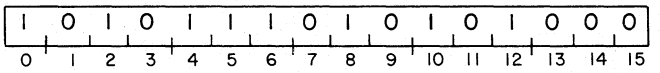
FSLE



If either the Z bit or the N bit in the FPSR is 1, the next sequential word is skipped.

SKIP ON GREATER THAN OR EQUAL TO ZERO

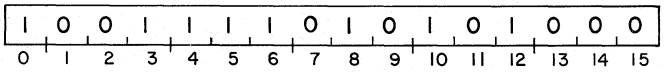
FSGE



If the N bit in the FPSR is 0, the next sequential word is skipped.

SKIP ON NON-ZERO

FSNE



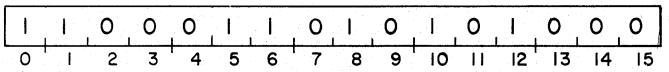
If the Z bit in the FPSR is 0, the next sequential word is skipped.

Error Test

There are eight instructions in the floating point instruction set that test the error indicators in the FPSR and skip on the result of the test. These instructions are described below.

SKIP ON NO MANTISSA OVERFLOW

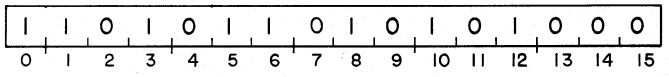
FSNM



If the mantissa overflow (MOF) bit in the FPSR is 0, the next sequential word is skipped.

SKIP ON NO UNDERFLOW

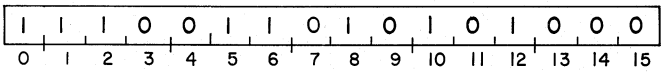
FSNU



If the underflow (UNF) bit in the FPSR is 0, the next sequential word is skipped.

SKIP ON NO OVERFLOW

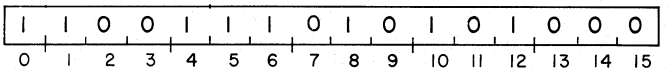
FSNO



If the overflow (OVF) bit in the FPSR is 0, the next sequential word is skipped.

SKIP ON NO ZERO DIVIDE

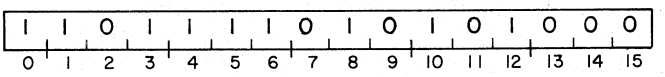
FSND



If the divide by zero (DVZ) bit in the FPSR is 0, the next sequential word is skipped.

SKIP ON NO UNDERFLOW AND NO ZERO DIVIDE

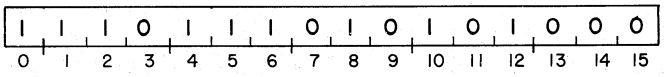
FSNUD



If both the underflow (UNF) bit and the divide by zero (DVZ) bit in the FPSR are 0, the next sequential word is skipped.

SKIP ON NO OVERFLOW AND NO ZERO DIVIDE

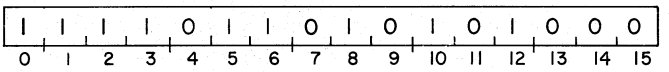
FSNOD



If both the overflow (OVF) bit and the divide by zero (DVZ) bit are 0, the next sequential word is skipped.

SKIP ON NO UNDERFLOW AND NO OVERFLOW

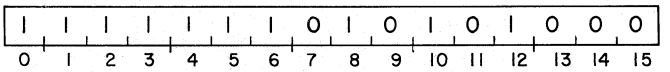
FSNUO



If both the underflow (UNF) bit and the overflow (OVF) bit in the FPSR are 0, the next sequential word is skipped.

SKIP ON NO ERROR

FSNER



If bits 1-4 in the FPSR are all 0, the next sequential word is skipped.

COMMERCIAL INSTRUCTION SET

An important feature of the ECLIPSE C/300 computer is its ability to perform operations on strings of characters and on decimal numbers. Instructions are included in this set that perform manipulations on data types commonly found in the commercial environment.

Commercial Faults

In the course of processing instructions in the commercial set, the CPU performs certain checks on the data being processed. If an invalid data type or number is found, a commercial fault is initiated. If a commercial fault is initiated, the processor places A code representing the type of fault in AC1, pushes a return block onto the stack with the program counter in the return block pointing to the instruction that caused the fault, and then executes a "jump indirect" to the commercial fault address. The codes placed in AC1 and their meanings are as follows:

CODE	MEANING
0	The EDIT instruction tried to process an invalid op-code.
1	An instruction was presented with an invalid data type.
2	An instruction was presented with an invalid sign character.
3	An instruction or EDIT op-code was presented with an invalid digit or character.
4	A LOAD INTEGER or STORE INTEGER instruction was presented with a number out of range.

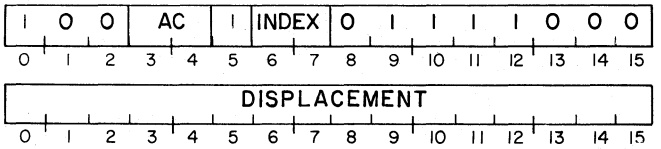
I/O Interrupts

Due to the potentially long time that may be required to perform any instruction in the commercial set in relation to I/O requests, all instructions in this set except for EXTENDED LOAD BYTE, EXTENDED STORE BYTE, and INTEGERIZE are interruptable. If a commercial instruction is interrupted, the program counter is decremented by one before it is placed in location 0 so that it points to the instruction in progress. All the commercial instructions maintain their operands in such a manner that any interrupt service routine that returns control to the interrupted program via the address stored in memory location 0 will correctly restart the interrupted instruction.

The processor assumes that no interrupt service program will alter the data being operated on by an interrupted instruction.

EXTENDED LOAD BYTE

ELDB ac, displacement<, index>



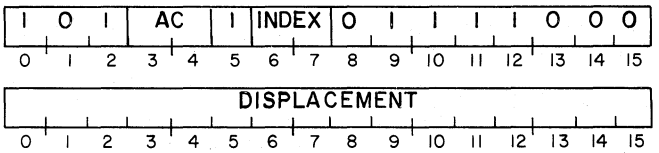
A byte pointer is formed by taking the index value, multiplying it by 2, and adding the low-order 16 bits of the result to the displacement. The byte addressed by this byte pointer is placed in bits 8-15 of the specified AC. Bits 0-7 of the specified AC are set to 0. Neither the index value nor the displacement are altered by the computation. The previous contents of the specified AC are lost.

The index value is computer from the index bits as follows:

INDEX BITS	INDEX VALUE
00	0
01	Address of the displacement field
10	Contents of AC2
11	Contents of AC3

EXTENDED STORE BYTE

ESTB ac, displacement<, index>



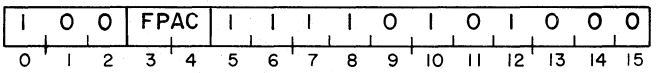
A byte pointer is formed by taking the index value, multiplying it by 2, and adding the low-order 16 bits of the result to the displacement. Bits 8-15 of the specified AC are placed in the byte addressed by this byte pointer. Neither the index value nor the displacement are altered by the computation. The contents of the specified AC remain unchanged.

The index value is computer from the index bits as follows:

INDEX BITS	INDEX VALUE
00	0
01	Address of the displacement field
10	Contents of AC2
11	Contents of AC3

LOAD INTEGER

LDI fpac



A decimal integer is converted to floating point form, normalized, and placed in the specified FPAC. The Z and N bits in the FPSR are set to reflect the new contents of the specified FPAC. The previous contents of the specified FPAC are lost.

AC1 must contain the attribute specifier word which describes the number.

AC3 must contain a byte pointer which is the address of the high-order byte of the number in memory.

Upon successful termination, the contents of AC0 and AC1 remain unchanged; AC2 contains the original contents of AC3; and AC3 is undefined.

This instruction will initiate a commercial fault under the following conditions:

1. For data types 0, 1, 2, 3, 4, and 5, if the instruction encounters an invalid digit or sign.
2. For data types 0, 1, 2, 3, 4, and 5, if the absolute value of the number is greater than $10^{16}-1$.
3. For data type 6, if the number is less than $-2^{56}-1$ or greater than 2^{56}.
4. For data type 7, if the size field is greater than 8.

In the event of a commercial fault, the contents of AC0 remain unchanged; AC1 contains the fault code; AC2 contains the original contents of AC3; and AC3 contains a byte pointer which is the address of the next byte to be processed.

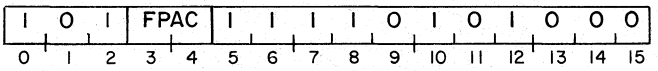
NOTES An attempt to load a minus 0 will result in the specified FPAC being set to true zero.

Numbers of data type 7 are not normalized after loading.

The first byte of numbers of data type 7 is assumed to contain the sign and exponent of the floating point number. The exponent must be in "excess 64" representation. The bytes following the first byte are loaded directly into the mantissa of the specified FPAC. Low-order bytes in the mantissa of the specified FPAC which do not receive bytes from memory are set to 0.

STORE INTEGER

STI fpac



The contents of the specified FPAC are converted to an integer of the specified type and stored, right-justified, in memory beginning at the specified location. The contents of the specified FPAC remain unchanged. The previous contents of the addressed memory locations are lost. The carry bit is set to 0. The condition codes in the FPSR remain unchanged.

AC1 must contain the attribute specifier word which describes the destination.

AC3 must contain a byte pointer which is the address of the high-order byte of the destination field in memory.

Upon successful termination, the contents of AC0 are undefined; AC1 remains unchanged; AC2 contains the original contents of AC3; and AC3 contains a byte pointer which is the address of the next byte after the destination field.

This instruction will initiate a commercial fault under the following condition: if the absolute value of the number contained in the specified FPAC is greater than 10^{16}. In the event of a commercial fault, the contents of AC0 are unchanged; AC1 contains the fault code; AC2 contains the original contents of AC3; the contents of AC3 are unpredictable; and the contents of the destination field are unpredictable.

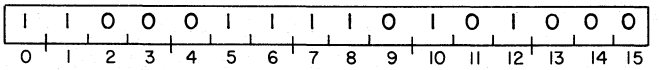
NOTES If the destination field cannot contain the entire number being stored, high-order digits are discarded until the number will fit into the destination. The remaining low-order digits are stored and the carry bit is set to 1.

For data types 0, 1, 2, 3, 4, 5, and 6, if the number being stored will not fill the destination field, the high-order bytes are set to 0.

For data type 7, if the number being stored will not fill the destination field, the low-order bytes are set to 0.

LOAD INTEGER EXTENDED

LDIX



A decimal integer of data type 0, 1, 2, 3, 4, or 5 is distributed into the four FPAC's. The integer is extended with high-order zeros until it is 32 digits long and then the low-order 8 digits are treated as an 8-digit number, converted to floating point form and placed in FAC3. The next 8 digits are treated as an 8-digit number, converted to floating point form and placed in FAC2. The next 8 digits are treated as an 8-digit number, converted to floating point form and placed in FAC1. The high-order 8 digits are treated as an 8-digit number, converted to floating point form and placed in FAC0. The sign of the integer is placed in each FPAC unless that FPAC has received 8 digits of zeros, in which case the FPAC is set to true zero. The Z and N bits in the FPSR are unpredictable.

AC1 must contain the attribute specifier which describes the integer.

AC3 must contain a byte pointer which is the address of the high-order byte of the integer.

Upon successful termination, the contents of AC0 and AC1 remain unchanged; AC2 contains the original contents of AC3; and AC3 is undefined.

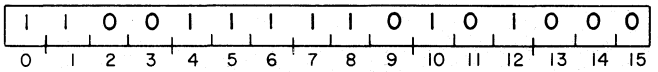
This instruction will initiate a commercial fault under the following conditions:

1. The attribute specifier word specifies data type 6 or 7.
2. The integer contains an invalid digit or sign.

In the event of a commercial fault, the contents of AC0 remain unchanged; AC1 contains the fault code; AC2 contains the original contents of AC3; and the contents of AC3 are unpredictable.

STORE INTEGER EXTENDED

STIX



The contents of the four FPAC's are converted to integer form and the low-order 8 digits of each are used to form a 32-digit integer. This integer is stored, right-justified, in memory beginning at the specified location. The sign of the integer is the logical OR of the signs of all four FPAC's. The previous contents of the addressed memory locations are lost. The carry bit is set to 0. The contents of the FPAC's remain unchanged. The condition codes in the FPSR are unpredictable.

AC1 must contain the attribute specifier word which describes the destination.

AC3 must contain a byte pointer which is the address of the high-order byte of the destination field in memory.

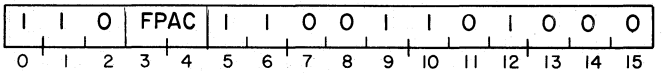
Upon successful termination, the contents of AC0 and AC1 remain unchanged; AC2 contains the original contents of AC3; and AC3 contains a byte pointer which is the address of the next byte after the destination field.

This instruction will initiate a commercial fault under the following condition: if the attribute specifier word specifies data type 6 or 7. In the event of a commercial fault, the contents of AC0 remain unchanged; AC1 contains the fault code; AC2 contains the original contents of AC3; the contents of AC3 are unpredictable; and the contents of the destination field remain unchanged.

NOTE If the destination field is not large enough to contain the number being stored, high-order digits are discarded until the number will fit in the destination. The low-order digits remaining are stored and the carry bit is set to 1.

INTEGERIZE

FINT

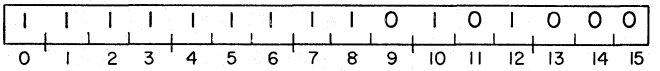


The number contained in the specified FPAC has its fractional portion (if any) set to 0 and then the number is normalized. The Z and N bits in the FPSR are set to reflect the new contents of the specified FPAC.

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.

LOAD SIGN

LSN



A number is evaluated and a code representing the value is placed in AC1. The value of the number and the resultant code is as follows:

VALUE	CODE
Positive non-zero	+1
Negative non-zero	-1
Positive zero	0
Negative zero	-2

AC1 must contain the attribute specifier word which describes the number.

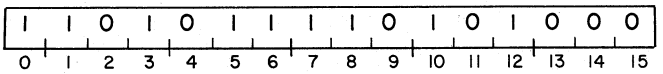
AC3 must contain a byte pointer which is the address of the high-order byte of the number.

Upon successful termination, the contents of AC0 remain unchanged; AC1 contains the value code; AC2 contains the original contents of AC3; and the contents of AC3 are unpredictable. The contents of the addressed memory locations remain unchanged.

This instruction will initiate a commercial fault under the following condition: if the instruction encounters an invalid digit or sign. In the event of a commercial fault, the contents of AC0 remain unchanged; AC1 contains the fault code; AC2 contains the original contents of AC3; the contents of AC3 are unpredictable; and the contents of the addressed memory locations remain unchanged.

CHARACTER MOVE

CMV



A number of bytes is fetched from one contiguous area of memory and stored into another contiguous area of memory under control of the values in the four accumulators. Fetching and storing may proceed from right to left or from left to right and may be in opposite directions. Moving continues until the destination field is filled. If the source field is longer than the destination field the carry bit is set to 1, otherwise it is set to 0. If the source field is shorter than the destination field, the destination field is padded with space characters.

AC0 must contain the number of bytes in the destination field. If this number is positive, the destination will be filled in ascending order, starting with the byte addressed by AC2. If this number is negative, the destination will be filled in descending order, starting with the byte addressed by AC2.

AC1 must contain the number of bytes in the source field. If this number is positive, the source bytes will be fetched in ascending order, starting with the byte addressed by AC3. If this number is negative, the source bytes will be fetched in descending order, starting with the byte addressed by AC3.

AC2 must contain a byte pointer which is the address of the first destination byte.

AC3 must contain a byte pointer which is the address of the first byte to be fetched.

The fields may overlap in any way.

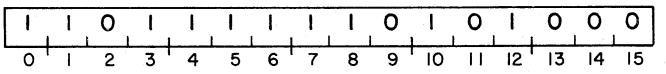
Upon termination, AC0 contains 0; AC1 contains the number of source bytes remaining to be fetched; AC2 contains a byte pointer which is the address of the next byte after the destination field; and AC3 contains a byte pointer which is the address of the next byte to be fetched.

NOTES If AC0 contains the number 0 at the beginning of this instruction, no bytes are fetched and none are stored.

If AC1 contains the number 0 at the beginning of this instruction, the destination field is filled with space characters.

CHARACTER COMPARE

CMP



One string of bytes is compared to another and a code reflecting the result is placed in AC1. The strings are processed one byte at a time and each byte is treated as an unsigned 8-bit binary quantity. If an inequality is found, the string possessing the lesser of the two bytes is considered the lesser string. The strings may be processed from left to right or from right to left and may be processed in opposite directions. If one string is shorter than the other, then, when that string is exhausted, it is treated as if it were padded with space characters to the length of the longer string. Comparison continues until an inequality is found or the longer string is exhausted. The contents of both strings remain unchanged. The result of the comparison and the corresponding code placed in AC1 is as follows:

RESULT	CODE
string 1 < string 2	-1
string 1 = string 2	0
string 1 > string 2	+1

AC0 must contain the number of bytes to be processed in string 2. If this number is positive, string 2 will be processed in ascending order, beginning with the byte addressed by AC2. If this number is negative, string 2 will be processed in descending order beginning with the byte addressed by AC2.

AC1 must contain the number of bytes to be processed in string 1. If this number is positive, string 1 will be processed in ascending order, beginning with the byte addressed by AC3. If this number is negative, string 1 will be processed in descending order beginning with the byte addressed by AC3.

AC2 must contain a byte pointer which is the address of the first byte to be processed in string 2.

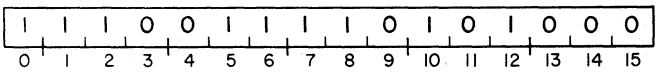
AC3 must contain a byte pointer which is the address of the first byte to be processed in string 1.

The fields may overlap in any way.

Upon termination, AC0 contains the number of bytes remaining to be processed in string 2; AC1 contains the return code; AC2 contains a byte pointer which is the address of either the failing byte in string 2 (if an inequality was found), or the next byte after string 2 (if string 2 was exhausted); and AC3 contains a byte pointer which is the address of either the failing byte in string 1 (if an inequality was found), or the next byte after string 1 (if string 1 was exhausted).

CHARACTER TRANSLATE

CTR



A string of bytes is translated from one data representation to another and either moved to another area of memory or compared to a second translated string. If the compare option is used, a code reflecting the result of the compare is placed in AC1. The strings are processed one byte at a time from left to right and processing continues until string 1 is exhausted. For the move option, the translated value of string 1 replaces string 2. For the compare option, the translated value of string 1 is compared to the translated value or string 2 on a byte for byte basis, treating both bytes as unsigned 8-bit binary quantities, until either an inequality is found or until string 1 is exhausted. If an inequality is found, the string possessing the lesser of the two bytes is considered the lesser string. For the move option, the contents of string 1 remain unchanged. For the compare option, the contents of both strings remain unchanged.

The translation is accomplished by treating each byte as an unsigned 8-bit binary integer and using that number as an index into a 256-byte translation table. The byte in the table addressed by using the source byte as an index is either stored in the next available byte of string 2 or is used in the compare.

For the compare option, the result of the comparison and the corresponding code placed in AC1 is as follows:

RESULT	CODE
Translated value of string 1 < translated value of string 2	-1
Translated value of string 1 = translated value of string 2	0
Translated value of string 1 > translated value of string 2	+1

AC0 must contain a word address of a word which contains a byte pointer which is the address of the first byte of the 256-byte translation table. If bit 0 of AC0 is set to 1, then the contents of AC0 are assumed to be the beginning of an indirection chain which will result in the address of a word which contains the byte pointer to the translation table.

AC1 must contain the number of bytes to be processed. Both strings will be processed in ascending order, beginning with the bytes addressed by AC2 and AC3. If the number in AC1 is negative, the move option is selected. If the number in AC1 is positive, the compare option is selected.

AC2 must contain a byte pointer which is the address of the first byte to be processed in string 2.

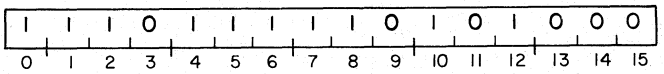
AC3 must contain a byte pointer which is the address of the first byte to be processed in string 1.

The fields may overlap in any way.

Upon termination of the instruction with the move option, AC0 contains the resolved address of the word which contains the byte pointer to the translation table; AC1 contains 0; AC2 contains a byte pointer which is the address of the next byte after string 2; and AC3 contains a byte pointer which is the address of the next byte after string 3. Upon termination of the instruction with the compare option, AC0 contains the resolved address of the word which contains the byte pointer to the translation table; AC1 contains the return code; AC2 contains a byte pointer which is the address of either the failing byte in string 2 (if an inequality was found) or the next byte after string 2 (if no inequality was found); and AC3 contains a byte pointer which is the address of either the failing byte in string 1 (if an inequality was found) or the next byte after string 1 (if no inequality was found).

CHARACTER MOVE UNTIL TRUE

CMT



A number of bytes is fetched from one contiguous area of memory and stored into another contiguous area of memory until either the source string is exhausted or until a specific character or one of a set of characters is encountered. The strings may be processed from left to right or from right to left, but both strings must be processed in the same direction. Each byte fetched from the source string is treated as an unsigned 8-bit binary integer and used as the bit index into a 256-bit table. If the addressed bit is 0, the byte is stored in the next available byte of the destination string and the next byte is fetched from the source string. If the addressed bit is 1, the byte is not stored and the instruction terminates. Processing continues until either the source string is exhausted or an addressed bit is 1.

AC0 must contain the word address of the first word of the 256-bit translation table. If bit 0 of AC0 is 1, the contents of AC0 are treated as the beginning of an indirection chain which will result in the word address of the first word of the translation table.

AC1 must contain the number of bytes to be processed. If the number is positive, processing will be in ascending order starting with the bytes addressed by AC2 and AC3. If the number is negative, processing will be in descending order starting with the bytes addressed by AC2 and AC3.

AC2 must contain a byte pointer which is the address of the first destination byte.

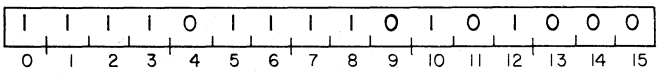
AC3 must contain a byte pointer which is the address of the first byte to be processed in the source string.

The fields may overlap in any way.

Upon termination, AC0 contains the resolved address of the translation table; AC1 contains the number of bytes that were not moved; AC2 contains a byte pointer which is the address of the next byte in the destination field; and AC3 contains a byte pointer which is the address of either the failing byte in the source string (if an addressed bit was 1) or the next byte after the source string (if no addressed bit was 1).

EDIT

EDIT



A decimal number is converted from either packed or unpacked form to a string of bytes under the control of an edit sub-program. This sub-program can perform many different operations on the number and its destination field including leading zero suppression, leading or trailing signs, floating fill characters, punctuation control, and insertion of text into the destination field.

Two indicators and three pointers are maintained by the EDIT instruction. The indicators are the Significance Trigger (T) and the Sign Flag (S). T is set to 1 when the first non-zero digit is processed unless otherwise specified by an edit op-code. At the beginning of an EDIT instruction, T is set to 0. S is set at the beginning of an EDIT to reflect the sign of the number being processed. If the number is positive, S is set to 0. If the number is negative, S is set to 1.

The three pointers are the Source Pointer (SI), the Destination Pointer (DI), and the op-code pointer (P). These pointers point to the current byte in process for the respective areas. At the beginning of an EDIT instruction, SI is set to the value contained in AC3, DI is set to the value contained in AC2, and P is set to the value contained in AC0.

The sub-program is made up of 8-bit op-codes followed by none, one, two, four, or several 8-bit operands. Op-codes are included for testing T and S; setting T and S; manipulating SI, DI, and P; and for moving bytes to and from areas in memory. The EDIT sub-program is processed sequentially, much the same way programs are processed. Unless instructed to do otherwise, P is updated after each operation to point to the next sequential op-code. The EDIT instruction will continue to process op-codes until directed to stop by the DEND op-code.

AC0 must contain a byte pointer which is the address of the first byte of the EDIT sub-program.

AC1 must contain the attribute specifier word which describes the number to be processed.

AC2 must contain a byte pointer which is the address of the first byte of the destination field.

AC3 must contain a byte pointer which is the address of the first byte of the source field.

The fields may overlap in any way.

Upon successful termination, the carry bit contains the significance trigger; AC0 contains a byte

pointer which is the address of the next op-code to be processed; the contents of AC1 are unpredictable; AC2 contains a byte pointer which is the address of the next destination byte; and AC3 contains a byte pointer which is the address of the next source byte.

This instruction will initiate a commercial fault under the following conditions:

1. If the attribute specifier word specifies data type 6 ot 7.
2. If the instruction encounters an invalid digit or sign.
3. If the instruction encounters an invalid edit op-code.
4. If the attribute specifier word specifies data type 5 and either the MOVE ALPHABETIC or the MOVE CHARACTER op-code is executed.

In the event of a commercial fault, AC0 contains a byte pointer which is the address of the op-code that failed plus 1 byte; AC1 contains the fault code; AC2 contains the current value of DI; and AC3 contains the current value of SI.

NOTES If SI is moved outside the area occupied by the source number, zeros will be supplied for numeric moves, even if SI is later moved back inside the source area.

The EDIT instruction places information on the stack. Therefore, the stack must be set up and have several words available for use.

If the EDIT instruction is interrupted, it places restart information on the stack and places 177777_8 in AC0.

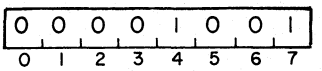
If the initial contents of AC0 are equal to 177777_8, the operation of the EDIT instruction is unpredictable.

In the description of some of the EDIT op-codes the symbol j is used to signify for how many characters a certain process is to take place. For those op-codes that use j, if the high-order bit of j is 1, then j is considered an 8-bit two's complement integer and is used to reference a word in the stack from which a 16-bit unsigned number is retrieved. This word is at the address (stack pointer +1+j). The number at this address is used instead of j for the remainder of that op-code.

The EDIT op-codes are described below.

SET T TO ONE

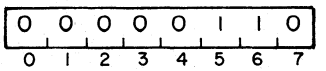
DSTO



The Significance Trigger (T) is set to 1.

SET T TO ZERO

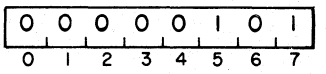
DSTZ



The Significance Trigger (T) is set to 0.

SET S TO ONE

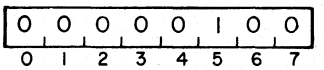
DSSO



The Sign Flag (S) is set to 1.

SET S TO ZERO

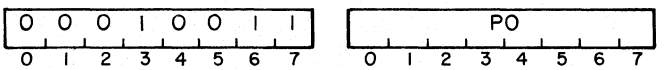
DSSZ



The Sign Flag (S) is set to 0.

ADD TO SI

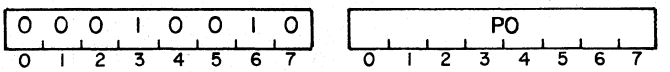
DASI <p0>



The 8-bit two's complement integer specified by p0 is added to the Source Indicator (SI).

ADD TO DI

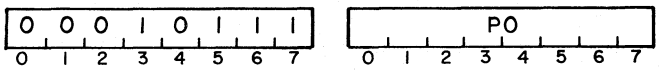
DADI <p0>



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

ADD TO P

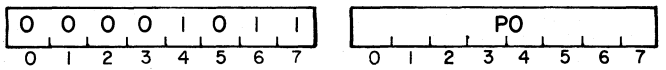
DAPU <p0>



The 8-bit two's complement integer specified by p0 is added to the op-code pointer (P). Before the add is performed, P is pointing to the byte containing the DAPU op-code.

ADD TO P DEPENDING ON T

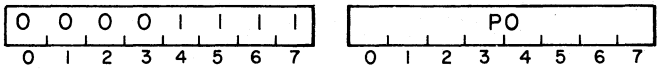
DAPT <p0>



If T is 1 the 8-bit two's complement integer specified by p0 is added to the op-code pointer (P). Before the add is performed, P is pointing to the byte containing the DAPT op-code.

ADD TO P DEPENDING ON S

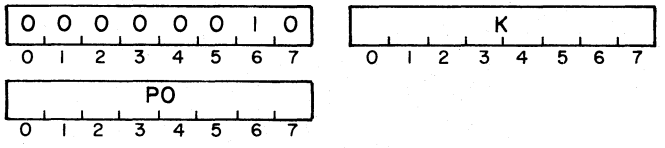
DAPS <p0>



If S is 0 the 8-bit two's complement integer specified by p0 is added to the op-code pointer (P). Before the add is performed, P is pointing to the byte containing the DAPS op-code.

STORE IN STACK

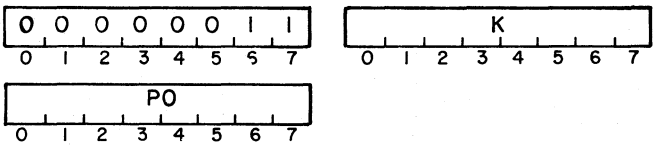
DSTK <k>,<p0>



The byte specified by p0 is stored in bits 8-15 of a word in the stack. Bits 0-7 of the word that receives p0 are set to 0. If the 8-bit two's complement integer specified by k is negative, the word that receives p0 is the word addressed by (stack pointer+1+k). If k is positive then p0 is stored at the address (frame pointer+1+k).

DECREMENT AND JUMP IF ZERO

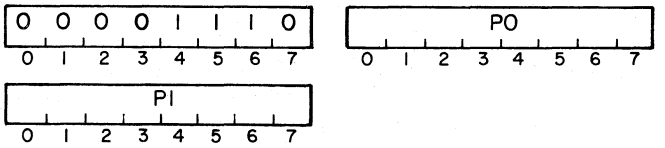
DDTK <k>,<p0>



A word in the stack is decremented by one. If the decremented value of the word is zero, the 8-bit two's complement integer specified by p0 is added to the op-code pointer (P). Before the add is performed, P is pointing to the byte containing the DDTK op-code. If the 8-bit two's complement integer specified by k is negative the word decremented is at the address (stack pointer+1+k). If k is positive, the word decremented is at the address (frame pointer+1+k).

INSERT SIGN

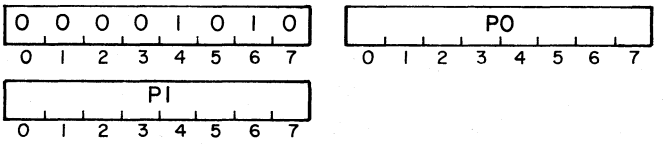
DINS <p0>,<p1>



If the Sign Flag (S) is 0 the character specified by p0 is inserted in the destination field at the position specified by DI. If S is 1 the character specified by p1 is inserted in the destination field at the position specified by DI. DI is incremented by 1.

INSERT CHARACTER SUPPRESS

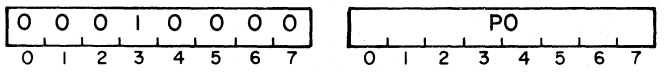
DINT <p0>,<p1>



If the Significance Trigger (T) is 0 the character specified by p0 is inserted in the destination field at the position specified by DI. If T is 1 the character specified by p1 is inserted in the destination field at the position specified by DI. DI is incremented by 1.

INSERT CHARACTER ONCE

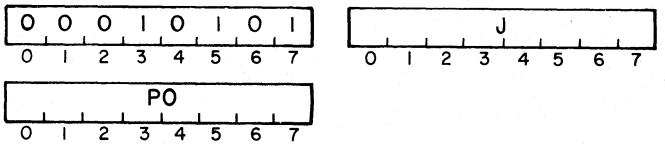
DINC <p0>



The character specified by p0 is inserted in the destination field at the position specified by DI. DI is incremented by one.

INSERT CHARACTER J TIMES

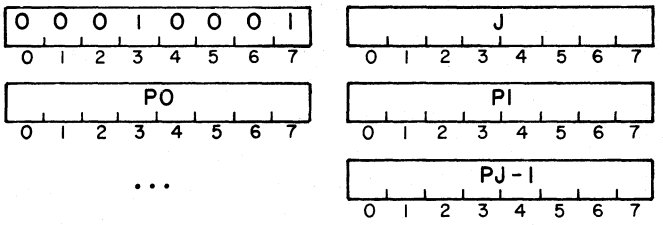
DIMC <j>,<p0>



The character specified by p0 is inserted into the destination field a number of times equal to j beginning at the position specified by DI. DI is increased by j.

INSERT CHARACTERS IMMEDIATE

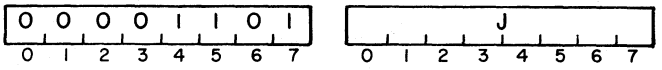
DICI <j>,<p0>,<p1>,...,<pj-1>



A number of characters equal to j is inserted from the op-code stream into the destination field beginning at the position specified by DI. Both DI and P are increased by j.

MOVE ALPHABETICS

DMVA <j>

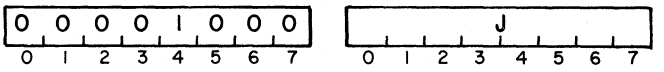


A number of alphabetic characters equal to j is moved from the source field beginning at the position specified by SI to the destination field beginning at the position specified by DI. Both SI and DI are increased by j. T is set to 1.

If the attribute specifier word indicates that the source field is data type 5 (packed), a commercial fault is initiated. If any of the characters moved is not an alphabetic (A-Z, a-z, or space), a commercial fault is initiated.

MOVE NUMERICS

DMVN <j>

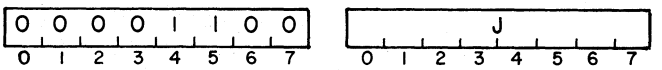


A number of numeric characters equal to j is moved from the source field beginning at the position specified by SI to the destination field beginning at the position specified by DI. Both SI and DI are increased by j. T is set to 1. In data type 2, if the least significant digit has been processed, SI is undefined.

If any of the characters moved is not a numeric (0-9 or space), a commercial fault is initiated.

MOVE CHARACTERS

DMVC <j>

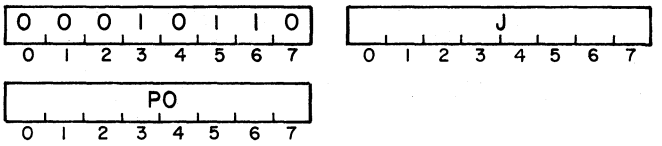


A number of characters equal to j is moved from the source field beginning at the position specified by SI to the destination field beginning at the position specified by DI. Both SI and DI are increased by j. T is set to 1.

If the attribute specifier word indicates that the source is data type 5 (packed), a commercial fault is initiated. No validation of the characters is performed.

MOVE NUMERIC WITH ZERO SUSPRESSION

DMVS <j>,<p0>

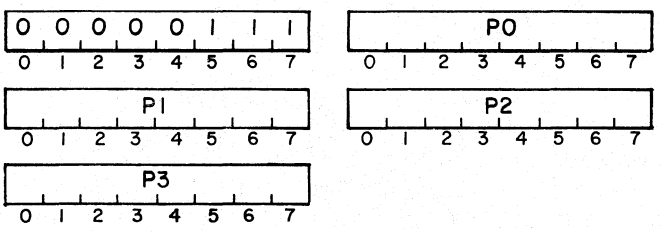


A number of numeric characters equal to j is moved from the source field beginning at the position specified by SI to the destination field beginning at the position specified by DI. If T is 1, the digit is moved from the source to the destination. As long as T is 0, all zeros and spaces are replaced with p0. When the first non-zero digit is encountered, T is set to 1. Both SI and DI are increased by j. In data type 2, if the least significant digit has been processed, SI is undefined.

If any of the characters moved is not a numeric (0-9 or space), a commercial fault is initiated.

MOVE DIGIT WITH OVERPUNCH

DMVO <p0>,<p1>,<p2>,<p3>



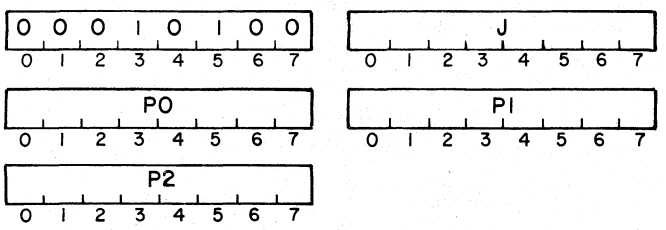
A digit substitute is placed in the destination field at the position specified by DI or a digit plus overpunch is moved from the source field at the position specified by SI to the destination field at the position specified by DI. Both SI and DI are increased by 1. In data type 2, if the least significant digit has been processed, SI is undefined.

If the digit is a zero or space then if S is 0 p0 is placed in the destination field; if S is 1 p1 is placed in the destination field. If the digit is a non-zero then if S is 0 p2 is added to the digit and the result is placed in the destination field; if S is 1 p3 is added to the digit and the result is placed in the destination field. If the digit is a non-zero T is set to 1.

If the character is not a numeric (0-9 or space) a commercial fault is initiated.

MOVE FLOAT

DMVF <j>,<p0>,<p1>,<p2>



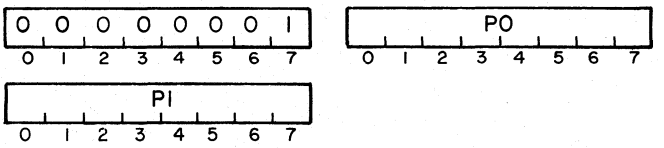
For a number of characters equal to j, either a digit substitute is placed in the destination field beginning at the position specified by DI or a digit is moved from the source field beginning at the position specified by SI to the destination field beginning at the position specified by DI. When T changes from 0 to 1, both the digit substitute and the digit are placed in the destination field. SI is increased by j. DI is increased by j if T does not change from 0 to 1 or by j+1 if T changes from 0 to 1. In data type 2, if the least significant digit has been processed, SI is undefined.

For each digit processed, if T is 1 the digit is moved from the source field to the destination field. If T is 0 and the digit is a zero or space p0 is placed in the destination field. If T is 0 and the digit is a non-zero then T is set to 1 and the characters placed in the destination field depend on S. If S is 0 p1 is placed in the destination field followed by the digit. If S is 1 p2 is placed in the destination field followed by the digit.

If any of the digits processed is not a numeric (0-9 or space) a commercial fault is initiated.

END FLOAT

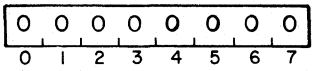
DNDF <p0>,<p1>



If T is 1 nothing is placed in the destination field and DI is left unchanged. If T is 0, then if S is 0 p0 is placed in the destination field at the position specified by DI. If S is 1 p1 is placed in the destination field at the position specified by DI. DI is increased by 1.

END EDIT

DEND



The EDIT sub-program is terminated.

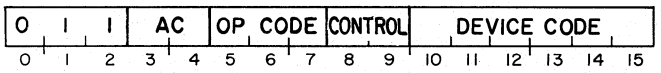
CHAPTER IV
INPUT/OUTPUT

INTRODUCTION

In order for the processor to perform useful work for the user, there must be some method for the program to transfer information outside the machine. The Input/Output (I/O) instruction set provides this facility. There are eight I/O instructions which allow the program to communicate with I/O devices, control the I/O interrupt system, control certain processor options, and to perform certain processor functions.

The ECLIPSE line has a 6-bit device selection network, corresponding to bits 10-15 in the I/O instruction format. Each device is connected to this network in such a way that each device will only respond to commands with its own device code. Each device also has two flags, Busy and Done, which control its operation. When Busy and Done are both zero, the device is idle and cannot perform any operations. To start a device, the program must set Busy to 1 and set Done to 0. When a device has finished its operation, it sets Busy to 0 and Done to 1.

The format for the I/O instructions is illustrated below.



Bits 0-2 are 011, bits 3-4 specify the AC, bits 5-7 contain the operation code, bits 8-9 control the Busy and Done flags in the device, and bits 10-15 specify the code of the device. The six bits provided for the device code in the I/O format mean that 64 unique device codes are available for use. Some of these device codes, however, are reserved for the CPU and certain processor options. The remaining device codes are available for referencing I/O units. Some of the codes have been assigned to specific devices by Data General and the assembler recognizes mnemonics for these devices. A complete listing of device codes, the devices assigned to these codes, and the mnemonics assigned to the devices is contained in Appendix A.

OPERATION OF I/O DEVICES

In general, the operation of all I/O devices is done by manipulating the Busy and Done flags. In order to operate a device, the program must first ensure that the device is not currently performing some operation. After the program has determined that the device is available, it can start an operation on the device by setting Busy to 1 and Done to 0. Once a device has completed its operation, and set Busy to 0 and Done to 1, it is available for another operation. The program can determine this condition in one of two ways. By using the I/O SKIP instruction, the program can test the status of the Busy and Done flags. Another way is to utilize the interrupt system that is standard on the ECLIPSE computer. The interrupt system is made up of an interrupt request line to which each I/O device is connected, an Interrupt On flag in the CPU, and a 16-bit interrupt priority mask. The Interrupt On flag controls the status of the interrupt system. If 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 respond to any interrupts. An interrupt request is initiated by an I/O device when it completes its operation. Upon completing the operation, the device sets Busy to 0 and Done to 1. At this time, the device also places an interrupt request on the interrupt request line, provided that the bit in the interrupt priority mask which corresponds to the priority level of the device is 0. If the mask bit is 1, the device sets Busy to 0 and Done to 1, but does not place an interrupt request on the interrupt request line.

If the Interrupt On flag is 1 at the time the processor completes execution of any instruction, the processor honors any requests on the interrupt request line. If the Interrupt On flag is 0, the CPU does not look at the interrupt request line; it just goes on to the next sequential instruction. The CPU honors an interrupt request by setting the Interrupt On flag to 0 so that no interrupts can interrupt the first part of the interrupt service routine. The CPU then places the updated program counter

into physical memory location 0 and executes a "JMP @1" instruction. It is assumed that physical location 1 contains the address, either direct or indirect, of the interrupt service routine.

Once the CPU has transferred control to the interrupt service routine, it is up to that routine to save any accumulators that will be used, save the carry bit if it will be used, determine which device requested the interrupt, and then service the interrupt. The determination of which device needs service can be done by I/O SKIP instructions or the routine can use the INTERRUPT ACKNOWLEDGE instruction. The saving of return information can be combined with the determination of which device is requesting service by use of the VECTOR ON INTERRUPTING DEVICE CODE instruction.

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 and uses the code returned as an index into a table of addresses. These addresses are the beginnings of the various device service routines. If more than one device is requesting service, the code returned is the code of that device requesting an interrupt which is physically closest to the CPU on the I/O bus. After servicing the device, the interrupt routine should restore all saved values, set the Interrupt On flag to 1, and return to the interrupted program. The instruction that sets the Interrupt On flag to 1 (INTERRUPT ENABLE) allows the processor to execute one more instruction (if the INTERRUPT ENABLE instruction changed the condition of the Interrupt On flag) before the next interrupt can take place. In order to prevent the interrupt service routine from going into a loop, this next instruction should be the instruction that returns control to the interrupted program. Since the updated value of the program counter was placed in location 0 by the CPU upon honoring the interrupt, all the interrupt routine has to do, after restoring the AC's and the carry bit, is execute an INTERRUPT ENABLE instruction and a "JMP @0" instruction and control will be returned to the interrupted program. If the main interrupt routine used the VECTOR instruction to save return information and to jump to the appropriate device service routine, then this information can be restored, and control returned to the interrupted program, by either the RESTORE or POP BLOCK instruction.

PRIORITY INTERRUPTS

If the Interrupt On flag remains 0 throughout the interrupt service routine, the interrupt routine cannot be interrupted and there is only one level of device priority. This level is determined by either the order in which the I/O SKIP instructions are issued or (if either INTERRUPT ACKNOWLEDGE or VECTOR are used) by the physical location of the devices on the bus. In a system with devices of widely differing speed, such as a teletypewriter versus a fixed head disc, the programmer may wish to set up a multiple level interrupt scheme. Hardware and instructions are available on the ECLIPSE line of computers to allow the implementation of sixteen levels of priority interrupts.

Each of the I/O devices is connected to a bit in the 16-bit priority mask. Devices which operate at roughly the same speed are connected to the same bit in the mask. Even though the standard mask bit assignments have the higher numbered bits assigned to lower speed devices, no implicit priority ordering is intended. The manner in which these priority levels are ordered is completely up to the programmer. The listing of device codes in Appendix A also contains the standard Data General mask bit assignments.

The condition of the priority mask is altered by the MASK OUT instruction. If a bit in the priority mask is set to 1, then all devices in the priority level corresponding to that bit will be prevented from requesting an interrupt when they complete an operation. In addition, all pending interrupt requests from devices in that priority level are disabled.

To implement a multiple priority level interrupt handler, the interrupt handler must be written in such a way that it may be interrupted without damage. For this to be possible, the main interrupt routine must save return information upon receiving control. The return information consists of the four accumulators, the carry bit, and the return address. This information should be stored in a unique place each time the interrupt handler is entered so that one level of interrupt does not overlay the return information that belongs to a lower priority level. The stack facility of the ECLIPSE computer and the VECTOR instruction allows this to be done in one instruction and stores the return information in a standard form. After saving the return information, the interrupt routine must determine which device requires service and jump to the correct service routine. This can be done in the same manner as for a single level interrupt handler. The VECTOR instruction does this at the same time that it is saving the return information.

After the correct service routine has received control, that routine should save the current priority mask, establish the new priority mask, and enable the interrupt system with the INTERRUPT ENABLE instruction. The VECTOR instruction does this in addition to its other operations. After servicing the interrupt, the routine should disable the interrupt system with the INTERRUPT DISABLE instruction, reset the priority mask, restore the accumulator, enable the interrupt system, and return control to the interrupted program. If the main interrupt handler uses the VECTOR instruction, then this dismissal process can be done by disabling the interrupt system, restoring the old priority mask, enabling the interrupt system and

then executing either a RESTORE or POP BLOCK instruction.

DATA CHANNEL

Handling data transfers between external devices and memory under program control requires the execution of several instructions for each word transferred. To allow greater transfer rates the ECLIPSE line of computers contains a data channel through which a device, at its own request, can gain direct access to memory using a minimum of processor time. At the maximum input rate of one word every 800ns or 1,250,000 words per second, or at the maximum output rate of one word every 1400ns or approximately 715,000 words per second, the data channel effectively stops the processor, but at lower rates, processing continues while data is being transferred.

When a device is ready to send or receive data, it requests access time via the channel. At the beginning of every memory cycle, the processor synchronizes any requests that are then being made. At certain specified points during the execution of an instruction, the CPU pauses to honor all previously synchronized requests. When a request is honored, a word is transferred directly via the channel from the device to memory or from memory to the device without specific action by the program. All requests are honored according to the relative position of the requesting devices on the I/O bus. That device requesting data channel service which is physically closest on the bus is serviced first, then the next closest device, and so on, until all requests have been honored. The synchronization of new requests occurs concurrently with the honoring of other requests. If a device continually requests the data channel, that device can prevent all devices further out on the bus from gaining access to the channel.

Following completion of an instruction, the processor handles all data channel requests, and then honors all outstanding I/O interrupt requests. After all data channel and I/O interrupt requests have been serviced, the processor continues with the next sequential instruction. The data channel is fully described in the "Programmer's Reference Manual for Peripherals" DGC 015-000021.

CODING AIDS

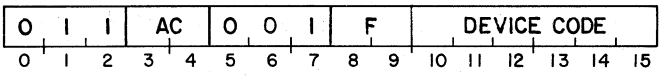
The set of I/O instructions has options that can be obtained by appending mnemonics to the standard mnemonics. These optional mnemonics and their result are given below.

CLASS ABBREVIATION	CODED CHARACTER	RESULT BITS	OPERATION
f	(omitted)	00	Does not affect the Busy and Done flags.
	S	01	Start the device by setting Busy to 1 and Done to 0.
	C	10	Idle the device by setting both Busy and Done to 0.
	P	11	Pulse the special in-out bus control line. The effect, if any, depends upon the device.

I/O INSTRUCTIONS

DATA IN A

DIA<f> ac, device

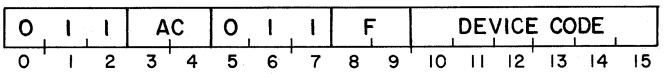


The contents of the A input buffer in the specified device are placed in the specified AC. After the data transfer, the Busy and Done flags are set according to the function specified by F.

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

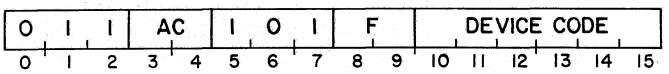


The contents of the B input buffer in the specified device are placed in the specified AC. After the data transfer, the Busy and Done flags are set according to the function specified by F.

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 C

DIC<f> ac, device

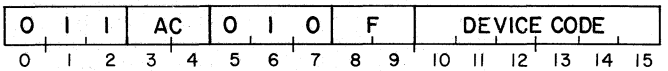


The contents of the C input buffer in the specified device are placed in the specified AC. After the data transfer, the Busy and Done flags are set according to the function specified by F.

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 OUT A

DOA<f> ac, device

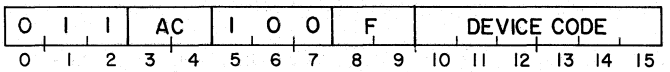


The contents of the specified AC are placed in the A output buffer of the specified device. After the data transfer, the Busy and Done flags are set according to the function specified by F. The contents of the specified AC remain unchanged.

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

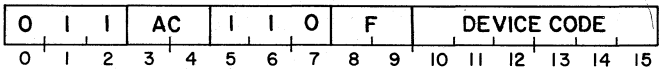


The contents of the specified AC are placed in the B output buffer of the specified device. After the data transfer, the Busy and Done flags are set according to the function specified by F. The contents of the specified AC remain unchanged.

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



The contents of the specified AC are placed in the C output buffer of the specified device. After the data transfer, the Busy and Done flags are set according to the function specified by F. The contents of the specified AC remain unchanged.

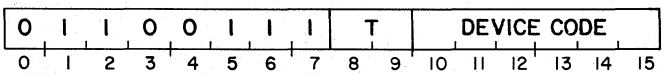
The number of data bits moved depends upon the size of the buffer and the mode of operation of the device.

The I/O SKIP instruction enables the programmer to make decisions based upon the values of the Busy and Done flags. Which test is performed is based upon the value of bits 8-9 in the instruction. Bits 8-9 can be set by appending an optional mnemonic to the I/O SKIP mnemonic. The optional mnemonics and their results are given below.

CLASS ABBREVIATION	CODED CHARACTER	RESULT BITS	OPERATION
t	BN	00	Tests for Busy = 1.
	BZ	01	Tests for Busy = 0.
	DN	10	Tests for Done = 1.
	DZ	11	Tests for Done = 0.

I/O SKIP

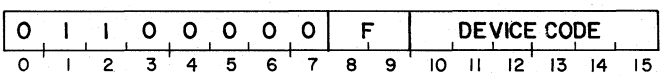
SKP<t> device



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

NO I/O TRANSFER

NIO<f> device



The Busy and Done flags in the specified device are set according to the function specified by F.

I/O instructions with a device code of 77_8 perform a number of special functions rather than controlling a specific device. In all but the I/O SKIP instruction, I/O instructions with a device code of 77_8 use bits 8-9 to control the condition of the Interrupt On flag. An I/O SKIP instruction with a device code of 77_8 uses bits 8-9 to either test the state of the Interrupt On flag or to test the state of the Power Fail flag. The mnemonics are the same as for normal I/O instructions. The table below gives the result of these bits for instructions with a device code of 77_8.

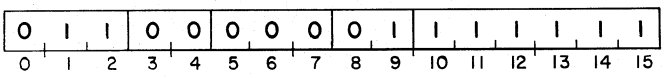
CLASS ABBREVIATION	CODED CHARACTER	RESULT BITS	OPERATION
f	(omitted)	00	Does not affect the state of the Interrupt On flag.
	S	01	Set the Interrupt On flag to 1.
	C	10	Set the Interrupt On flag to 0.
	P	11	Does not affect the state of the Interrupt On flag. Used only in the VCT instruction.

The device code of 77_8 deals mainly with processor functions and has, therefore, been given the mnemonic of CPU. In addition, many of the I/O instructions that reference this device code have been given special mnemonics. While these special mnemonics are functionally equivalent to the corresponding I/O instructions with a device code of 77_8, there is the following limitation; the mnemonics for controlling the state of the Interrupt On flag cannot be appended to them. If the programmer wishes to alter the state of the Interrupt On flag while performing a MASK OUT instruction, for example, he must issue the appropriate I/O instruction (DOB<f> ac, CPU) instead of the corresponding special mnemonic (MSKO ac). If the special mnemonic is used, bits 8-9 are set to 00. In describing the instructions, the special mnemonic for the corresponding I/O instruction will be given first, followed by the I/O instruction.

INTERRUPT ENABLE

INTEN

NIOS CPU



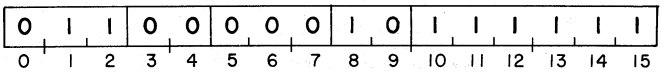
The Interrupt On flag is set to 1.

If the state of the Interrupt On flag is changed by this instruction, the CPU allows one more instruction to execute before the first I/O interrupt can occur. However, if the instruction is one of those that is interruptable, then interrupts can occur as soon as the instruction begins to execute.

INTERRUPT DISABLE

INTDS

NIOC CPU

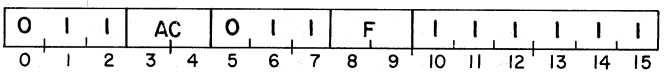


The Interrupt On flag is set to 0.

INTERRUPT ACKNOWLEDGE

INTA ac

DIB<f> ac, CPU

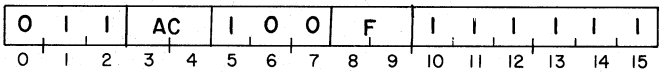


The six-bit device code of that device requesting an interrupt which is physically closest to the CPU on the bus is placed in bits 10-15 of the specified AC. Bits 0-9 of the specified AC are set to 0. After the transfer, the Interrupt On flag is set according to the function specified by F.

MASK OUT

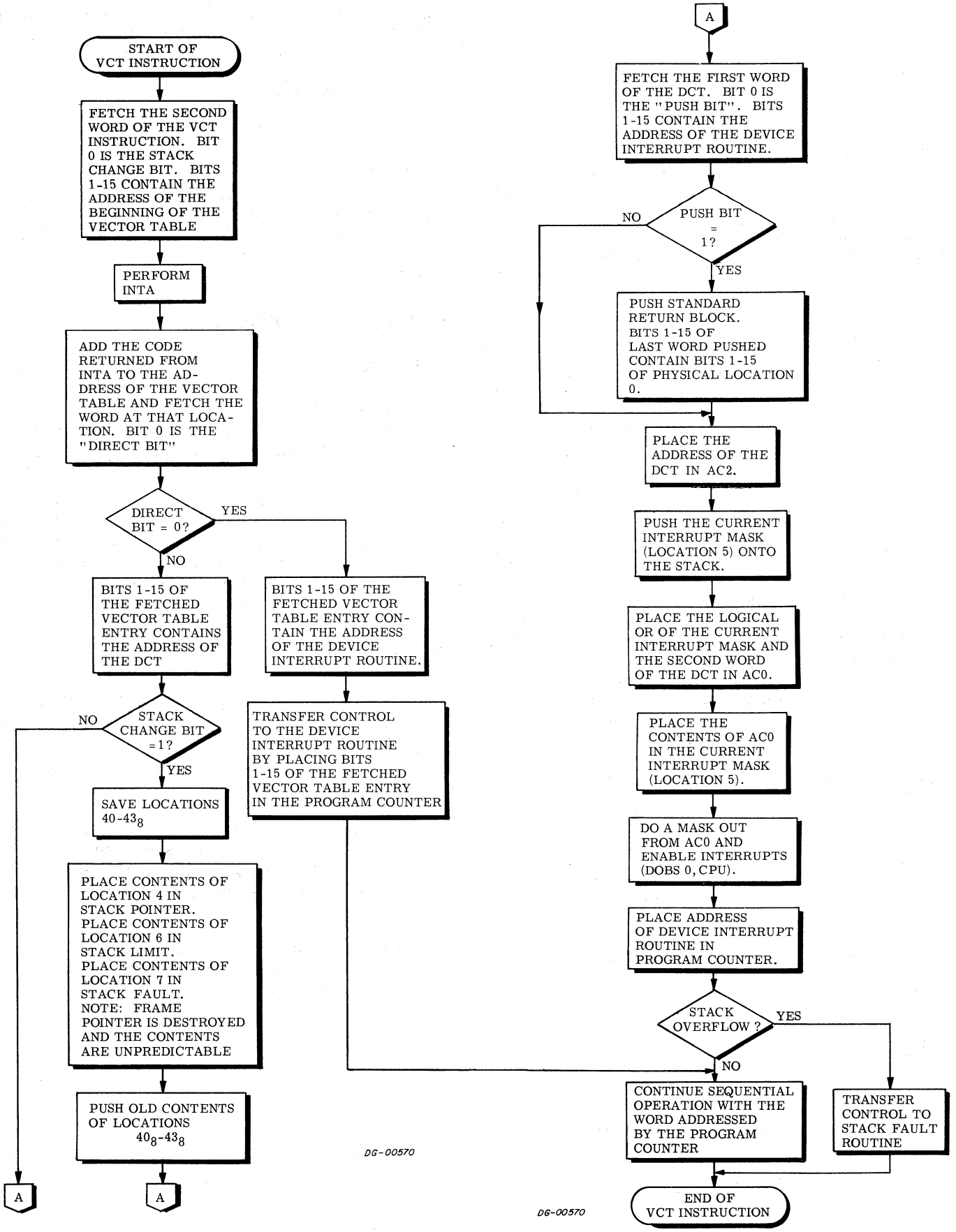
MSKO ac

DOB<f> ac, CPU



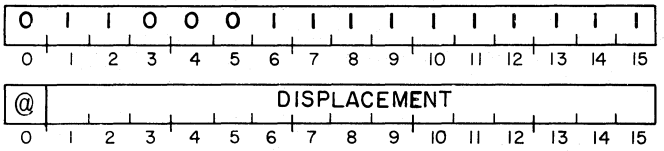
The contents of the specified AC are placed in the priority mask. After the transfer, the Interrupt On flag is set according to the function specified by F. The contents of the specified AC remain unchanged.

NOTE A 1 in any bit disables interrupt requests from devices in the corresponding priority level.


START OF VCT INSTRUCTION
FETCH THE SECOND WORD OF THE VCT INSTRUCTION. BIT 0 IS THE STACK CHANGE BIT. BITS 1-15 CONTAIN THE ADDRESS OF THE BEGINNING OF THE VECTOR TABLE
PERFORM INTA
ADD THE CODE RETURNED FROM INTA TO THE ADDRESS OF THE VECTOR TABLE AND FETCH THE WORD AT THAT LOCATION. BIT 0 IS THE "DIRECT BIT"
DIRECT BIT = 0?
YES
NO
BITS 1-15 OF THE FETCHED VECTOR TABLE ENTRY CONTAINS THE ADDRESS OF THE DCT
BITS 1-15 OF THE FETCHED VECTOR TABLE ENTRY CONTAIN THE ADDRESS OF THE DEVICE INTERRUPT ROUTINE.
STACK CHANGE BIT =1?
NO
YES
SAVE LOCATIONS 40-43₈
TRANSFER CONTROL TO THE DEVICE INTERRUPT ROUTINE BY PLACING BITS 1-15 OF THE FETCHED VECTOR TABLE ENTRY IN THE PROGRAM COUNTER
PLACE CONTENTS OF LOCATION 4 IN STACK POINTER. PLACE CONTENTS OF LOCATION 6 IN STACK LIMIT. PLACE CONTENTS OF LOCATION 7 IN STACK FAULT. NOTE: FRAME POINTER IS DESTROYED AND THE CONTENTS ARE UNPREDICTABLE
PUSH OLD CONTENTS OF LOCATIONS 40₈-43₈
A
A
DG-00570
A
FETCH THE FIRST WORD OF THE DCT. BIT 0 IS THE "PUSH BIT". BITS 1-15 CONTAIN THE ADDRESS OF THE DEVICE INTERRUPT ROUTINE.
NO
PUSH BIT = 1?
YES
PUSH STANDARD RETURN BLOCK. BITS 1-15 OF LAST WORD PUSHED CONTAIN BITS 1-15 OF PHYSICAL LOCATION 0.
PLACE THE ADDRESS OF THE DCT IN AC2.
PUSH THE CURRENT INTERRUPT MASK (LOCATION 5) ONTO THE STACK.
PLACE THE LOGICAL OR OF THE CURRENT INTERRUPT MASK AND THE SECOND WORD OF THE DCT IN AC0.
PLACE THE CONTENTS OF AC0 IN THE CURRENT INTERRUPT MASK (LOCATION 5).
DO A MASK OUT FROM AC0 AND ENABLE INTERRUPTS (DOBS 0, CPU).
PLACE ADDRESS OF DEVICE INTERRUPT ROUTINE IN PROGRAM COUNTER.
STACK OVERFLOW ?
YES
NO
CONTINUE SEQUENTIAL OPERATION WITH THE WORD ADDRESSED BY THE PROGRAM COUNTER
TRANSFER CONTROL TO STACK FAULT ROUTINE
END OF VCT INSTRUCTION
DG-00570

VECTOR ON INTERRUPTING DEVICE CODE

VCT <@>displacement



This instruction provides a fast and efficient method for transferring control from the main I/O interrupt handler to the correct interrupt service routine for the interrupting device. Bit 0 of the second word of the instruction is the "stack change bit" and bits 1-15 contain the address of a 64-word vector table. Vector table entries are one word in length and consist of a "direct" bit in bit 0 followed by an address in bits 1-15.

An INTERRUPT ACKNOWLEDGE instruction is performed. The device code returned is added to the address of the vector table and the vector table entry at that address is fetched. If the direct bit in the fetched vector table entry is 0, the address in bits 1-15 is taken to be the address of the device handler routine for the interrupting device and control is immediately transferred there by placing the address in the program counter.

If the direct bit is 1, the address in bits 1-15 of the vector table entry is taken to be the address of the device control table (DCT) for the interrupting device. At this point, the stack change bit is examined. If the stack change bit is 0, no stack change is performed. If the stack change bit is 1, a new stack is created by placing the contents of memory location 6 in the stack limit, and the contents of memory location 7 in the stack fault. The previous contents of memory locations 40_8-43_8 are then pushed onto this new stack.

Device control tables must consist of at least two words. The first word of a DCT consists of a "push bit" in bit 0 followed by the address of the device handler routine for the interrupting device in bits 1-15. The second word of a DCT contains a mask that will be used to construct the new interrupt priority mask. Succeeding words in a DCT may contain information that is to be used by the device interrupt handler.

After the stack change procedure is performed, the first word of the DCT is fetched and inspected. If the push bit is 1, a standard return block is pushed onto the stack with bits 1-15 of physical location 0 placed in bits 1-15 of the last word pushed. If the push bit is 0, no return block is pushed.

Following this procedure, the address of the DCT is placed in bits 1-15 of AC2 and bit 0 of AC2 is set to 0.

Next, the current interrupt priority mask is pushed on the stack. The contents of the second word of DCT are logically OR'd with the current interrupt priority mask and the result is placed in both AC0 and memory location 5. This constructs the new interrupt priority mask and places it in AC0 and the save location for the mask. A DOBS 0,CPU instruction is now performed. This is a MASK OUT instruction that also enables the interrupt system.

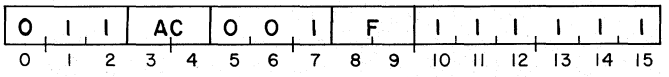
After a new interrupt priority mask is established and the interrupt system enabled, control is transferred to the device handler by placing bits 1-15 of the first word of the DCT in the program counter.

A detailed discussion of the use of the VECTOR instruction and its impact on interrupt latency can be found in Appendix G.

READ SWITCHES

READS ac

DIA<f> ac,CPU

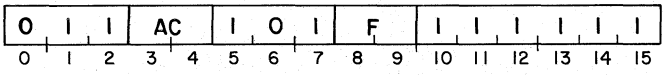


The setting of the console data switches is placed in the specified AC. After the transfer, the Interrupt On flag is set according to the function specified by F.

I/O RESET

IORST

DIC<f> ac,CPU



The Busy and Done flags in all I/O devices are set to 0. The 16-bit priority mask is set to 0. The Interrupt On flag is set according to the function specified by F.

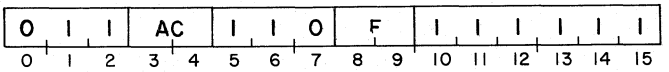
NOTE The assembler recognizes the instruction IORST as equivalent to DICC 0,CPU.

If the mnemonic DIC is used to perform this function, an accumulator must be coded to avoid assembly errors. Regardless of how the instruction is coded, during execution, the AC field is ignored and the contents of the AC remain unchanged.

HALT

HALTA ac

DOC<f> ac,CPU



The Interrupt On flag is set according to the function specified by F and then the processor is stopped. The data lights display the contents of the specified AC.

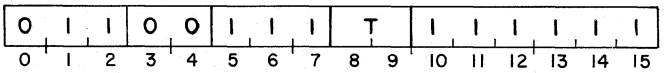
NOTE The assembler recognizes the instruction HALT as equivalent to HALTA 0.

The CPU SKIP instruction enables the programmer to make decisions based upon the value of the Interrupt On flag or the Power Fail flag. Which test is performed is based upon the value of bits 8-9 in the instruction. Bits 8-9 can be set by appending an optional mnemonic to the CPU SKIP mnemonic. The optional mnemonics and their results are given below.

CLASS ABBREVIATION	CODED CHARACTER	RESULT BITS	OPERATION
t	BN	00	Tests for Interrupt On = 1.
	BZ	01	Tests for Interrupt On = 0.
	DN	10	Tests for Power Fail = 1.
	DZ	11	Tests for Power Fail = 0.

CPU SKIP

SKP<t> CPU



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

ERROR CHECKING AND CORRECTION

The Error Checking and Correction (ERCC) feature is designed for applications where either a high degree of reliability is required for the main memory of a system, or where a graceful "fail-soft" capability is desired in the event of memory errors. The ERCC feature will detect and correct all single-bit errors that occur in memories equipped with the option.

The ERCC feature is a combination processor and memory feature in that parts are present in both to provide the facility. The ERCC feature is available with any memory available with the ECLIPSE line of computers.

Method of Operation

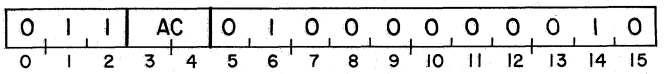
The word length of an ERCC memory is 21 bits. These 21 bits are broken into 16 data bits followed by 5 ERCC bits (COR0-COR4). This check field is constructed by a hardware encoder from the 16 data bits and is written each time the memory location is written into. When the memory location is read, the encoder recomputes the ERCC bits read from memory. If the computed bits match the bits read from memory, the 16 data bits are passed on to the CPU. If the bits do not match, a single bit error has occurred. The memory pauses while the single bit in error is corrected and the entire corrected word is rewritten into the memory location. The data is then passed on to the CPU and the ERCC option requests an interrupt. If no error occurs, no time is taken and the cycle time of the memory is unchanged from its non-ERCC counterpart.

The logic of the ERCC feature is such that all single-bit errors are detected and corrected. In the rare event that a multi-bit error occurs, either it is detected and reported as such with no correction, or it is incorrectly interpreted as a single-bit error and that bit is complemented.

The operation of the ERCC option is governed by one I/O instruction. Two other instructions are used to interrogate the option after it has detected and corrected an error. The ERCC option has no Busy flag and no mask bit in the priority mask. The device code for the ERCC option is 2. The instructions for the ERCC option are described below.

ENABLE ERCC

DOA ac, ERCC



The ERCC option is enabled according to the setting of bits 14-15 of the specified AC. Bits 0-13 of the specified AC are ignored. The contents of the specified AC remain unchanged. The bit settings and their meanings are described below.

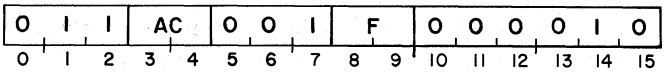
BIT SETTING	MEANING
00	Disable checking and correction. Write valid check field.
01	Disable checking and correction. For core memory, write check field of 11111. For semiconductor memory, do not alter the check field.
10	Enable checking and correction. Do not interrupt on memory error.
11	Enable checking and correction. Interrupt on memory error.

After Power Up or I/O reset, the ERCC option is in the 10 state.

NOTE When the ERCC feature detects and corrects a memory error, it sets its Done flag to 1. The Done flag will remain 1 until the ERCC feature receives a Start pulse or an I/O RESET instruction is issued. Receipt of a Start pulse will also set the fault address to 0.

READ MEMORY FAULT ADDRESS

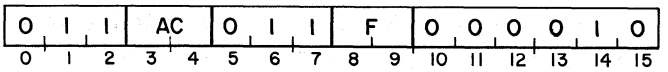
DIA<f> ac, ERCC



The low-order 16 bits of the 17-bit physical address of the memory location in error is placed in bits 0-15 of the specified AC. The previous contents of the specified AC are lost.

READ MEMORY FAULT CODE

DIB<f> ac, ERCC



A five bit error code is placed in bits 0-4 of the specified AC. Bits 5-14 of the specified AC are set to 0. The high-order bit of the 17-bit physical address of the failing location is placed in bit 15. These codes tell which bit was in error and has been corrected. The codes and their meanings are described below.

ERCC Feature Memory Fault Codes

FAULT CODE	MEANING
00000	No error.
00001	The error was in check bit 4.
00010	The error was in check bit 3.
00011	The error was in data bit 0.
00100	The error was in check bit 2.
00101	The error was in data bit 1.
00110	The error was a multiple bit error.
00111	The error was in data bit 3.
01000	The error was in check bit 1.
01001	The error was in data bit 4.
01010	All 21 bits of the memory location are 1.
01011	The error was in data bit 6.
01100	The error was in data bit 7.
01101	The error was in data bit 8.
01110	The error was in data bit 9.
01111	The error was a multiple bit error.
10000	The error was in check bit 0.
10001	The error was in data bit 11.
10010	The error was in data bit 12.
10011	The error was in data bit 13.
10100	The error was in data bit 14.
10101	All 21 bits of the memory location are 0.
10110	The error was in data bit 2.
10111	The error was a multiple bit error.
11000	The error was in data bit 10.
11001	The error was a multiple bit error.
11010	The error was in data bit 5.
11011	The error was a multiple bit error.
11100	The error was in data bit 15.
11101	The error was a multiple bit error.
11110	The error was a multiple bit error.
11111	The error was a multiple bit error.

REAL TIME CLOCK

The Real Time Clock (RTC) feature of the ECLIPSE line of computers generates a sequence of pulses that is independent of the CPU timing. It will generate I/O interrupts at any one of four program selectable frequencies. The Busy and Done flags of the RTC option are controlled by bits 8-9 of the I/O instruction. The RTC option is device code 14_8 and has the mnemonic RTC. The interrupt disable bit is priority mask bit 13.

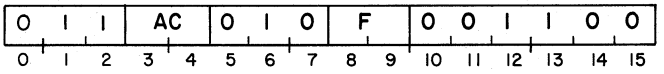
Setting Busy allows the next pulse from the clock to set Done, and the RTC option requests an I/O interrupt if its priority mask bit is 0. A SELECT RTC FREQUENCY instruction to select the clock frequency only has to be given once. After each interrupt, an NIOS instruction will set up the clock for the next interrupt.

When Busy is first set the first interrupt can come at any time up to the clock period. After the first interrupt has occurred, succeeding interrupts come at the clock frequency, provided that the program always sets Busy before the clock period expires. After power up or I/O reset, the clock is set to the line frequency. After power up the line frequency pulses are available immediately, but five seconds must elapse before a steady pulse train is available from the clock for other frequencies.

The RTC frequency is selected by the following instruction.

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.

AC bits 14-15	Frequency
00	ac line frequency
01	10Hz
10	100Hz
11	1000Hz

POWER FAIL/AUTO-RESTART

In the ECLIPSE line of computers, when power is turned off and then on again, core memory is unaltered. However, when the power is turned on, the state of the accumulators, the program counter, and the various flags in the CPU is indeterminate. The power fail option provides a "fail-soft" capability in the event of unexpected power loss.

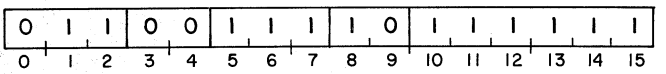
In the event of power failure, there is a delay of one to two milliseconds before the processor shuts down. The power fail option senses the imminent loss of power, sets the Power Fail flag, and requests an interrupt. The interrupt service routine can then use this delay to store the contents of the 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 on the ECLIPSE computer, so there is more than enough time to perform the power fail routine.

When power is restored, the action taken by the automatic restart portion of the power fail option depends upon the position of the power switch on the front panel. If the switch is in the "on" position, the CPU remains stopped after power is restored. If the switch is in the "lock" position, then 222ms after power is restored, the CPU executes a "jump indirect" to location 0, restarting the interrupted program.

The power fail option has no priority mask bit in the priority mask. It does not respond to the INTERRUPT ACKNOWLEDGE instruction. It responds to the VECTOR instruction with device code 0. Testing of the Power Fail flag by the CPU SKIP instruction is described below.

SKIP IF POWER FAIL FLAG IS ONE

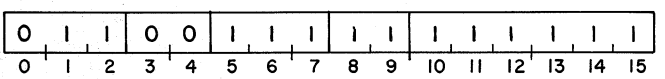
SKPDN CPU



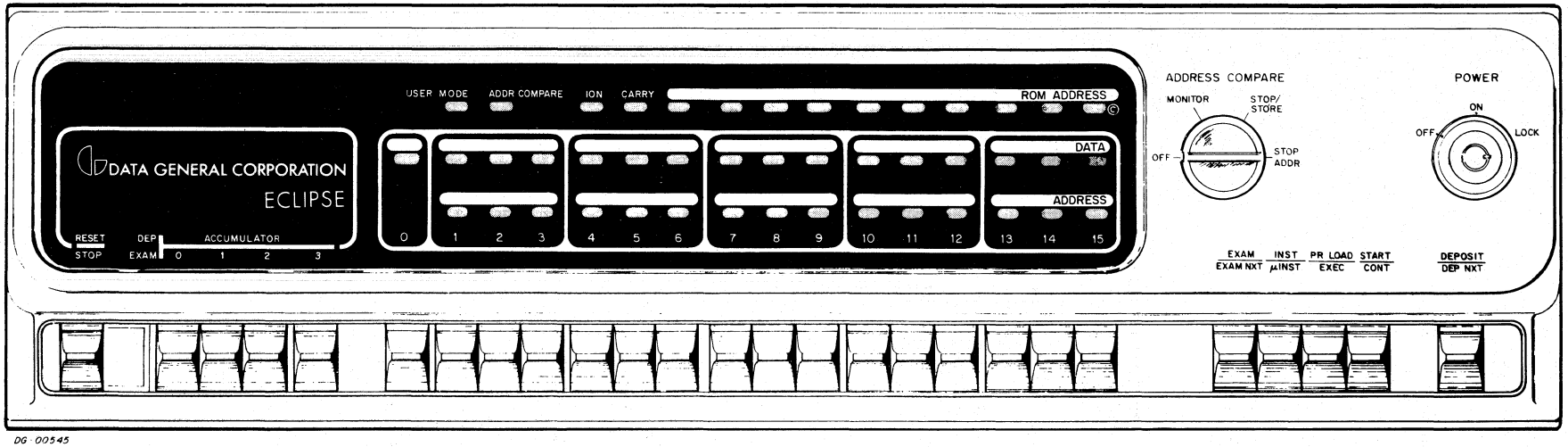
If the Power Fail flag is 1 (i.e., power is failing), the next sequential word is skipped.

SKIP IF POWER FAIL FLAG IS ZERO

SKPDZ CPU



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



DG 00545

ECLIPSE LINE Computer Console

CHAPTER V
FRONT PANEL

INTRODUCTION

The front panel of the ECLIPSE line of computers contains all the functions' switches and displays all the information needed to operate the machine. The function and data switches allow the operator to perform many useful operations and the lights reflect the current state of the machine. If a light is lit, it means the corresponding bit is 1. If the light is not lit, the corresponding bit is 0. The lights and their meanings are described below.

LIGHT	MEANING WHEN LIT
USER MODE	The MAP feature is translating addresses in the user mode.
ADDR COMPARE	Operation of the machine is suspended because the comparison requested by the ADDRESS COMPARE switch has come up true.
ION	The Interrupt On flag is 1.
CARRY	The carry bit is 1.
ROM ADDRESS	These ten lights display the address in the micro-code of the next micro-instruction to be fetched.
DATA	These 16 lights display what is currently in general register 0 of the micro-code processor.
ADDRESS	These 15 lights display what is currently in the memory address bus.

CONSOLE SWITCHES

In a row along the bottom of the console are 26 switches. These are broken down into three groups; 5 function switches, 16 data switches, and 5 more function switches. The ten function switches are spring loaded. When pushed up, they perform one function, when pushed down, they perform another function. When released, these switches return to a neutral "off" position. The 16 data switches are two-position toggle switches. When in the up position, they represent a 1; when in the down position, they represent a 0. These switches have no neutral position. These 16 switches can be used to enter either data or addresses. If the switches are to be interpreted as data, all 16 data switches are used and they correspond to the bits in an internal 16-bit word. The leftmost switch of this group corresponds to bit 0 and the rightmost switch corresponds to bit 15. If the switches are to be interpreted as an address, only the rightmost 15 switches are used. When interpreted as an address, the second switch from the left is the high-order bit of the address and the rightmost switch is the low-order bit. All addresses coming from the console are treated as logical addresses.

Starting from the left of the console and proceeding to the right, the function switches and their meanings are described below.

Reset-Stop

When this switch is pushed up, the RESET function is performed and an I/O RESET instruction is executed. The CPU is stopped after completing the current processor cycle. The Interrupt On flag, the 16-bit priority mask, and all Busy and Done flags are set to 0. While in this state, the CPU will honor data channel requests.

When this switch is pushed down, the STOP function is performed. The CPU is stopped after completing the current instruction and before executing the next instruction. If an I/O device requests an interrupt during the execution of the current instruction, it is not honored before the CPU is stopped. All outstanding data channel requests are honored before the CPU is stopped. Data channel requests are continually honored while the machine is in the stopped state. After the CPU is stopped, the address lights display the

address of the next instruction to be executed. The contents of the data lights are unpredictable.

Deposit-Examine

The next four switches are the accumulator DEPOSIT-EXAMINE switches. The switches are numbered 0-3 from left to right. Each switch affects only its corresponding accumulator. When one of these switches is pushed up, the current setting of the data switches is deposited into the corresponding accumulator. The data lights display the information placed in the AC.

When one of these switches is pushed down, the contents of the corresponding accumulator are displayed in the data lights.

Exam-Exam Nxt

When this switch is pushed up, the EXAMINE function is performed. The address indicated by data switches 1-15 is placed in the program counter. This value is displayed in the address lights. The contents of the word addressed by the program counter are then read and displayed in the data lights.

When this switch is pushed down, the EXAMINE NEXT function is performed. The current value of the program counter is incremented by one and the new value is displayed in the address lights. The contents of the word addressed by the updated value of the program counter are then read and displayed in the data lights.

Inst-μ/Inst

When this switch is pushed up, the INSTRUCTION STEP function is performed. The instruction contained in the word addressed by the current value of the program counter is executed and then the CPU is stopped. The address lights display the updated value of the program counter. The contents of the data lights are unpredictable.

> **NOTE** If the machine is stopped while in the user mode and the LOAD EFFECTIVE ADDRESS instruction is enabled for the current user, and a LOAD EFFECTIVE ADDRESS instruction is executed by use of the instruction step function, the action of the console is undefined.

When this switch is pushed down, the MICRO-INSTRUCTION STEP function is performed. The next micro-instruction in logical sequence is performed and the micro-code processor is stopped. The ROM address lights display the micro-code address of the next microinstruction to be fetched. The address lights display the contents of the memory address bus, and the data lights display the contents of the memory bus for the microinstruction just performed.

PR Load-Exec

When this switch is pushed up, the program load function is performed. The contents of the bootstrap read-only memory are placed in memory locations $0\text{-}37_8$ and a "JMP 0" instruction is performed.

When this switch is pushed down, the EXECUTE function is performed. The current setting of the data switches is interpreted as an instruction and that instruction is executed as if it were in memory at the location specified by the program counter. After the instruction is stopped, the address lights display the updated value of the program counter. The contents of the data lights are unpredictable.

> **NOTE** If the machine is stopped while in the user mode and the LOAD EFFECTIVE ADDRESS instruction is enabled for the current user, and a LOAD EFFECTIVE ADDRESS instruction is executed by use of the execute function, the action of the console is undefined.

Start-Cont

When this switch is pushed up, the START function is performed. The address indicated by data switches 1-15 is placed in the program counter and sequential operation of the processor begins with the word addressed by the updated value of the program counter.

When this switch is pushed down, the CONTINUE function is performed. Sequential operation of the processor continues from the current state of the machine.

Dep-Dep Next

When this switch is pushed up, the DEPOSIT function is performed. The current setting of the data switches is placed into the word addressed by the current value of the program counter. The updated value of the altered word is displayed in the data lights.

When this switch is pushed down, the DEPOSIT NEXT function is performed. The program counter is incremented by one and the current setting of the data switches is placed into the word addressed

by the updated value of the program counter. The updated value of the program counter is displayed in the address lights and the updated value of the altered word is displayed in the data lights.

Address Compare

The ADDRESS COMPARE switch is a four position rotary switch. The four positions are labeled "OFF", "MONITOR", STOP/STORE", and "STOP/ADDR". The functions of these four positions are described below.

Off

When the switch is in the OFF position, the ADDRESS COMPARE feature is disabled.

Monitor

When the switch is in the MONITOR position, it is possible to examine and monitor locations in memory while the CPU is running. When the switch is in this position, the contents of the memory location addressed by the current setting of the data switches is displayed in the data lights each time the location is accessed by the CPU. The data is not displayed until either the CPU accesses the location or the EXAM-EXAM NXT switch is pushed up. The data lights continue to display this information until either the contents of the addressed location are altered by the CPU or the setting of the data switches is changed. In the first case, the updated value of the location is displayed in the data lights. In the second case, the old data remains in the lights until either the CPU accesses the location addressed by the new data switch setting or the EXAM-EXAM NXT switch is pushed up. As soon as the CPU accesses the location addressed by the new switch setting or the EXAM-EXAM NXT switch is pushed up, the contents of the location addressed by the new switch setting will be displayed in the data lights.

Stop/Store

With the switch in the STOP/STORE position, the ADDRESS COMPARE feature will suspend the operation of the CPU if the CPU tries to alter the location whose address is set in the data switches. The addressed location is altered. The ADDR COMPARE light is lit to indicate that the ADDRESS COMPARE feature has suspended the operation of the machine. The contents of the data and address lights are unpredictable.

Stop/Addr

With the switch in the STOP/ADDR position, the ADDRESS COMPARE feature will suspend the operation of the CPU if the CPU tries to access the location whose address is set in the data switches. The addressed location is neither read nor written. The ADDR COMPARE light is lit to indicate that the ADDRESS COMPARE feature has suspended the operation of the machine. The contents of the data and address lights are unpredictable.

Power

The POWER switch is a three position key switch. The three positions are labeled "OFF", "ON", and "LOCK". With the switch in the OFF position, all power to the CPU is shut off and the machine will not run. Turning the switch to the ON position turns on the power, performs a RESET function, and enables all the switches. Turning the switch to the LOCK position allows the key to be removed. While the switch is in the LOCK position, all console functions except the MONITOR function of the ADDRESS COMPARE feature are disabled.

PROGRAM LOADING

Before a program can be executed it must be brought into memory. This requires that a loading program already reside in memory. If the memory does not contain a loading program, the operator can either enter a bootstrap loader into memory via the data switches or he can use the "PROGRAM LOAD" feature. Pushing the PR LOAD-EXEC switch on the console to the up position deposits a 32_{10} word bootstrap loader into the first 32_{10} memory locations and then begins sequential operation at memory location 0. This bootstrap loader will then read in a loader program from an I/O device. This bootstrap loader can use either programmed I/O to read in a loader from a low-speed device such as the teletypewriter or paper tape reader, or data channel transfers to read in a loader from a high-speed device such as magnetic tape or disc.

To enter a loader program, the operator must first set up the device that is to be used and set its octal device code into data switches 10-15. If the device is a data channel device, set data switch 0 to 1. If the device is not a data channel device, set data switch 0 to 0. After this is done, push the PR LOAD-EXEC switch to the up position. The bootstrap loader will be deposited into memory locations $0\text{-}37_8$ and started at location 0.

The bootstrap loader reads the data switches, sets up its own I/O instructions with the specified device code, and then performs a program load procedure depending upon the state of data switch 0.

If the switch is a 1, the bootstrap loader starts the device for data channel storage beginning at location 0 and then loops at location 377_8 until a data channel transfer places a word into that location.

NOTE For proper program loading via the data channel, the device used must be initiated for reading by an I/O RESET followed by an NIOS instruction. In addition, it is up to the device to stop reading after 256 words have been read.

After a word has been placed in location 377_8, it is executed as an instruction. Typically, this word is either a HALT or a JUMP into the data that the data channel has placed in the first 377_8 memory locations.

If data switch 0 is a 0, the bootstrap loader reads the loader program via programmed I/O. The device must supply 8-bit data bytes, and each pair of bytes is stored as a single word in memory, wherein the first and second bytes read become the left and right halves of the word. To simplify the positioning of the tape in the reader, the bootstrap loader ignores leading null characters. It does not begin storing any words until it reads a non-zero synchronization byte. The first word following this synchronization byte must be the negative of the total number of words to be read, including the first word. The number of words to be read, including the first word may not be greater than 192_{10}. The bootstrap loader stores these words beginning at memory location 100_8. After storing the last word read, it transfers control to that location.

Listed below is the standard 32 word bootstrap loader for the ECLIPSE line of computers. This program is capable of loading in either of the manners described above.

The usual procedure is to use the bootstrap loader to bring in a larger program that sizes memory and then reads in the binary loader, storing it at the top of memory.

BOOTSTRAP LOADER

```
BEG:    IORST
        READS   0           ;READ SWITCHES INTO AC0
        LDA     1,C77       ;GET DEVICE MASK (000077)
        AND     0,1         ;ISOLATE DEVICE CODE
        COM     1,1         ;- DEVICE CODE - 1
LOOP:   ISZ     OP1         ;COUNT DEVICE CODE INTO ALL
        ISZ     OP2         ;I/O INSTRUCTIONS
        ISZ     OP3
        INC     1,1,SZR     ;DONE?
        JMP     LOOP        ;NO, INCREMENT AGAIN
        LDA     2,C377      ;YES, PUT JMP 377 INTO LOCATION 377
        STA     2,377
OP1:    060077              ;START DEVICE: (NIOS 0) - 1
        MOVL    0,0,SZC     ;LOW SPEED DEVICE? (TEST SWITCH 0)
C377:   JMP     377         ;NO, GO TO 377 AND WAIT FOR CHANNEL
LOOP2:  JSR     GET+1       ;GET A FRAME
        MOVC    0,0,SNR     ;IS IT NON-ZERO?
        JMP     LOOP2       ;NO, IGNORE AND GET ANOTHER
LOOP4:  JSR     GET         ;YES, GET FULL WORD
        STA     1,@C77      ;STORE STARTING AT 100 2's COMPLEMENT OF WORD COUNT
                            ; (AUTOINCREMENT)
        ISZ     100         ;COUNT WORD - DONE?
        JMP     LOOP4       ;NO, GET ANOTHER
C77:    JMP     77          ;YES - LOCATION COUNTER AND JUMP TO LAST WORD
GET:    SUBZ    1,1         ;CLEAR AC1, SET CARRY
OP2:
LOOP3:  063577              ;DONE?: (SKPDN 0) - 1
        JMP     LOOP3       ;NO, WAIT
OP3:    060477              ;YES, READ IN AC0: (DIAS 0,0) - 1
        ADDCS   0,1,SNC     ;ADD 2 FRAMES SWAPPED - GOT SECOND?
        JMP     LOOP3       ;NO, GO BACK AFTER IT
        MOVS    1,1         ;YES, SWAP THEM
        JMP     0,3         ;RETURN WITH FULL WORD
        0                   ;PADDING
```

This page intentionally left blank.

APPENDICES

- I/O DEVICE CODES
- OCTAL AND HEXADECIMAL CONVERSION
- ASCII CHARACTER CODES
- DOUBLE PRECISION ARITHMETIC
- COMPATIBILITY WITH NOVA LINE COMPUTERS
- INSTRUCTION EXECUTION TIMES
- USE OF THE VECTOR INSTRUCTION
- INSTRUCTION USE EXAMPLES

APPENDIX A
I/O DEVICE CODES

OCTAL DEVICE CODE	MNEMONIC	PRIORITY MASK BIT	DEVICE NAME
00	----	--	Unused
01	WCS	--	Writeable control store
02	ERCC	--	Error checking and correction
03	MAP	--	Memory allocation and protection
04			
05			
06	MCAT	12	Multiprocessor adapter transmitter
07	MCAR	12	Multiprocessor adapter receiver
10	TTI	14	TTY input
11	TTO	15	TTY output
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	--	D/A converter
24	DCM	0	Data communications multiplexor
25			
26			
27			
30	QTY	14	Asynchronous hardware multiplexor
30	SLA	14	Synchronous line adapter
31[2]	IBM1	13	IBM 360/370 interface
32	IBM2		
33	DKP	7	Moving head disc
34	CAS	10	Cassette tape
34[2]	MX1	11	Multiline asynchronous controller
35	MX2		
36	IPB	6	Interprocessor bus--half duplex
37	IVT	6	IPB watchdog timer
40	DPI	8	IPB full duplex input

[2]Code returned by INTA and used by VCT

APPENDIX A (Continued)

I/O DEVICE CODES

OCTAL DEVICE CODES	MNEMONIC	PRIORITY MASK BIT	DEVICE NAME
41	DPO	8	IPB full duplex output
40[3]	SCR	8	Synchronous communication receiver
41[4]	SCT	8	Synchronous communication transmitter
42	DIO	7	Digital I/O
43	DIOT	6	Digital I/O timer
44	MXM	12	Modem control for MX1/MX2
45			
46	MCAT1	12	Second multiprocessor transmitter
47	MCAR1	12	Second multiprocessor receiver
50	TTI1	14	Second TTY input
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 disc
61	ADCV1	8	Second A/D converter
62	MTA1	10	Second magnetic tape
63	DACV1	--	Second D/A converter
64			
65			
66			
67			
70	QTY1	14	Second asynchronous hardware multiplexor
70	SLA1	14	Second synchronous line adapter
71[2]		13	Second IBM 360/370 interface
72			
73	DKP1	7	Second moving head disc
74	CAS1	10	Second cassette tape
74[2]		11	Second multiline asynchronous controller
75			
76			
77	CPU	--	Central processor and console functions

[2]Code returned by INTA and used by VCT

[3]Can be set up with any unused even device code equal to 40 or above

[4]Can be set up with any unused odd device code equal to 41 or above

APPENDIX B

OCTAL AND HEXADECIMAL CONVERSION

To convert a number from octal or hexadecimal to decimal, locate in each column of the appropriate table the decimal equivalent for the octal or hex digit in that position. Add the decimal equivalents to obtain the decimal number.

To convert a decimal number to octal or hexadecimal:

1. Locate the largest decimal value in the appropriate table that will fit into the decimal number to be converted;
2. note its octal or hex equivalent and column position;
3. find the decimal remainder.

Repeat the process on each remainder. When the remainder is 0, all digits will have been generated.

	8^5	8^4	8^3	8^2	8^1	8^0
0	0	0	0	0	0	0
1	32,768	4,096	512	64	8	1
2	65,536	8,192	1,024	128	16	2
3	98,304	12,228	1,536	192	24	3
4	131,072	16,384	2,048	256	32	4
5	163,840	20,480	2,560	320	40	5
6	196,608	24,576	3,072	384	48	6
7	229,376	28,672	3,584	448	56	7

	16^5	16^4	16^3	16^2	16^1	16^0
0	0	0	0	0	0	0
1	1,048,576	65,536	4,096	256	16	1
2	2,097,152	131,072	8,192	512	32	2
3	3,145,728	196,608	12,288	768	48	3
4	4,194,304	262,144	16,384	1,024	64	4
5	5,242,880	327,680	20,480	1,280	80	5
6	6,291,456	393,216	24,576	1,536	96	6
7	7,340,032	458,752	28,672	1,792	112	7
8	8,388,608	524,288	32,768	2,048	128	8
9	9,437,184	589,824	36,864	2,304	144	9
A	10,485,760	655,360	40,960	2,560	160	10
B	11,534,336	720,896	45,056	2,816	176	11
C	12,582,912	786,432	49,152	3,072	192	12
D	13,631,488	851,968	53,248	3,328	208	13
E	14,680,064	917,504	57,344	3,584	224	14
F	15,728,640	983,040	61,440	3,840	240	15

APPENDIX C
ASCII CHARACTER CODES

Decimal	7-bit Octal	ASCII Character	Control Function	To Produce On TTY Mod 33, 35 Cntrl	Shift	Char	Even Parity 8-bit code
0	000	NUL	Null	✓	✓	P	000
1	001	SOH	Start of Heading	✓		A	201
2	002	STX	Start of Text	✓		B	202
3	003	ETX	End of Text	✓		C	003
4	004	EOT	End of Transmission	✓		D	204
5	005	ENQ	Enquiry	✓		E	005
6	006	ACK	Acknowledge	✓		F	006
7	007	BEL	Bell	✓		G	207
8	010	BS	Backspace	✓		H	210
9	011	HT	Horizontal Tab	✓		I	011
10	012	NL	New Line			line feed	012
				✓		J	012
				✓		line feed	212[1]
11	013	VT	Vertical Tab	✓		K	213
12	014	FF	Form Feed	✓		L	014
13	015	RT	Return			return	215
				✓		M	215
				✓		return	015[1]
14	016	SO	Shift Out	✓		N	216
15	017	SI	Shift In	✓		O	017
16	020	DLE	Data Link Escape	✓		P	220
17	021	DC1	Device Control 1	✓		Q	021
18	022	DC2	Device Control 2	✓		R	022
19	023	DC3	Device Control 3	✓		S	223
20	024	DC4	Device Control 4	✓		T	024
21	025	NAK	Negative Acknowledge	✓		U	225
22	026	SYN	Synchronous Idle	✓		V	226
23	027	ETB	End Transmission Block	✓		W	027
24	030	CAN	Cancel	✓		X	030
25	031	EM	End of Medium	✓		Y	231
26	032	SUB	Substitute	✓		Z	232
27	033	ESC	Escape			esc	033
				✓	✓	K	033
28	034	FS	File Separator	✓	✓	L	234
29	035	GS	Group Separator	✓	✓	M	035
30	036	RS	Record Separator	✓	✓	N	036
31	037	US	Unit Separator	✓	✓	O	237
32	040	SP	Space			space	240
33	041	!			✓	1	041
34	042	"			✓	2	042
35	043	#			✓	3	243
36	044	$			✓	4	044
37	045	%			✓	5	245
38	046	&			✓	6	246
39	047	'			✓	7	047
40	050	(			✓	8	050
41	051	)			✓	9	251

[1] On even parity TTY's, these codes are odd parity.

APPENDIX C (Continued)

ASCII CHARACTER CODES

Decimal	7-bit Octal	Character	To Produce On TTY Mod 33, 35 Cntrl Shift Char	Even Parity 8-bit Code
42	052	*	✓ :	252
43	053	+	✓ ;	053
44	054	,	,	
45	055	-	-	055
46	056	.	.	056
47	057	/	/	257
48	060	0	0	060
49	061	1	1	261
50	062	2	2	262
51	063	3	3	063
52	064	4	4	264
53	065	5	5	065
54	066	6	6	066
55	067	7	7	267
56	070	8	8	270
57	071	9	9	071
58	072	:	:	072
59	073	;	;	273
60	074	<	✓ ,	074
61	075	=	✓ -	275
62	076	>	✓ .	276
63	077	?	✓ /	077
64	100	@	✓ P	300
65	101	A	A	101
66	102	B	B	102
67	103	C	C	303
68	104	D	D	104
69	105	E	E	305
70	106	F	F	306
71	107	G	G	107
72	110	H	H	110
73	111	I	I	311
74	112	J	J	312
75	113	K	K	113
76	114	L	L	314
77	115	M	M	115
78	116	N	N	116
79	117	O	O	317
80	120	P	P	120
81	121	Q	Q	321
82	122	R	R	322
83	123	S	S	123
84	124	T	T	324

APPENDIX C (Continued)

ASCII CHARACTER CODES

Decimal	7-bit Octal	Character	To Produce On TTY Mod 33, 35 Cntrl Shift Char	Even Parity 8-bit Code
85	125	U	U	125
86	126	V	V	126
87	127	W	W	327
88	130	X	X	330
89	131	Y	Y	131
90	132	Z	Z	132
91	133	[	↙ K	333
92	134	\	↙ L	134
93	135	]	↙ M	335
94	136	^	↙ N	336
95	137	–	↙ 0	137
96	140	`		140
97	141	a		341
98	142	b		342
99	143	c		143
100	144	d		344
101	145	e		145
102	146	f		146
103	147	g		347
104	150	h		350
105	151	i		151
106	152	j		152
107	153	k		353
108	154	l		154
109	155	m		355
110	156	n		356
111	157	o		157
112	160	p		360
113	161	q		161
114	162	r		162
115	163	s		363
116	164	t		164
117	165	u		365
118	166	v		366
119	167	w		167
120	170	x		170
121	171	y		371
122	172	z		372
123	173	{		173
124	174	|		374
125	175	}		175
126	176	~		176
127	177	DEL	rubout	377

APPENDIX D
DOUBLE PRECISION ARITHMETIC

A double length number consists of two words concatenated into a 32-bit string wherein bit 0 is the sign and bits 1-31 are the magnitude in two's complement notation. The high-order part of a negative number is therefore in one's complement form unless the low order part is null (at the right, only 0's are null regardless of sign). Hence, in processing double length numbers, two's complement operations are usually confined to the low order parts, whereas one's complement operations are generally required for the high-order parts.

Suppose we wish to negate the double length number whose high and low order words respectively are in AC0 and AC1. We negate the low order part, but we simply complement the high-order part unless the low order part is zero. Hence

```
NEG 1,1,SNR
NEG 0,0,SKP ;LOW ORDER ZERO
COM 0,0     ;LOW ORDER NONZERO
```

Note that the magnitude parts of the sequence of negative numbers from the most negative toward zero are the positive numbers from zero upward. Hence, in multiple precision arithmetic, low-order words can be treated simply as positive numbers. In unsigned addition a carry indicates that the low-order result is just too large and the high-order part must be increased. We add the number in AC2 and AC3 to the number in AC0 and AC1.

```
ADDZ 3,1,SZC
INC  0,0
ADD  2,0
```

In two's complement subtraction a carry should occur unless the subtrahend is too large. We could increment as in addition, but since incrementing in the high-order part is precisely the difference between a one's complement and a two's complement, we can always manage with only two instructions. We subtract the number in AC2 and AC3 from that in AC0 and AC1.

```
SUBZ 3,1,SZC
SUB  2,0,SKP
ADC  2,0
```

APPENDIX E

COMPATIBILITY WITH NOVA LINE COMPUTERS

The ECLIPSE line of computers is compatible with the Data General NOVA line of computers. Any program presently running on any NOVA line computer will run on an ECLIPSE line computer without change provided that it does not violate any of the following constraints:

1. The program may not be dependent on instruction execution times or Input/Output (I/O) transfer times. Times for the ECLIPSE line of computers may be faster than a NOVA line computer depending upon the application.
2. The program may not use any fixed-point arithmetic instructions that have both the "no-load" and "no-skip" options specified. The ECLIPSE line of computers uses these codes to implement instructions in the standard instruction set.
3. The program may not require the hardware multiply/divide option available on any NOVA line computer.
4. The program may not utilize the data channel increment or add-to-memory features.
5. The program may not utilize either the memory management and protection option or the hardware floating point option currently available for NOVA line computers.
6. The memory and I/O resources available on the ECLIPSE line computer should be at least equivalent to those available on the NOVA line computer for which the program was designed.

A violation of the third constraint can be easily corrected. The multiply and divide available in the ECLIPSE line of computers standard instruction set are functionally equivalent to the operations provided in the hardware multiply/divide option for the NOVA line computers. Only the operation codes must be changed to take advantage of the ECLIPSE line of computers multiply and divide. Similarly, only small changes need be made to a program which uses the current NOVA line floating point option in order for that program to take advantage of the ECLIPSE line of computers floating point option. The floating point number formats are the same.

APPENDIX F

INSTRUCTION EXECUTION TIMES

The following table gives minimum, maximum, and typical execution times for all instructions in the basic instruction set. These times assume a system without the MAP feature operating with 4-way interleaved core memory. All times are in microseconds.

STANDARD INSTRUCTION SET	MINIMUM	MAXIMUM	TYPICAL	NOTES
ADD	0.6	0.6	0.6	1
ADD COMPLEMENT	0.6	0.6	0.6	1
ADD IMMEDIATE	0.6	0.6	0.6	
EXTENDED ADD IMMEDIATE	1.2	1.2	1.2	
AND	0.6	0.6	0.6	1
AND IMMEDIATE	1.2	1.2	1.2	
AND WITH COMPLEMENTED SOURCE	0.6	0.6	0.6	
BLOCK ADD AND MOVE	1.8+0.8N	1.8+1.2N	1.7 +1.0N	2
BLOCK MOVE	2.0+0.6N	1.4+1.2N	1.85+0.85N	2
COMPARE LIMITS	specified AC's not the same			
number within limits	1.8	2.2	1.9	
number less than L	2.0	2.4	2.1	
number greater than H	2.2	2.6	2.3	
	specified AC' s the same			
number within limits	1.6	1.6	1.6	
number less than L	1.8	1.8	1.8	
number greater than H	1.6	1.6	1.6	
COMPLEMENT	0.6	0.6	0.6	1
COUNT BITS	1.0	10.6	1.0 +0.6N	3
DECIMAL ADD	0.6	0.6	0.6	
DECREMENT AND SKIP IF ZERO	1.4	1.6	1.5	4
EXTENDED DECREMENT AND SKIP IF ZERO	2.4	2.6	2.45	4
DECIMAL SUBTRACT	0.6	0.6	0.6	
DISPATCH				
number within limits	3.6	3.6	3.6	4,11
number less than L	2.8	2.8	2.8	4
number greater than H	2.6	2.8	2.65	
DOUBLE HEX SHIFT LEFT	2.4	4.2	depends on shift count	
DOUBLE HEX SHIFT RIGHT	2.4	4.2	depends on shift count	
DOUBLE LOGICAL SHIFT	1.0	5.6	depends on shift count	
ENTER WCS	depends on user instruction			
EXCHANGE ACCUMULATORS	0.8	0.8	0.8	
EXCLUSIVE OR	0.6	0.6	0.6	
EXCLUSIVE OR IMMEDIATE	1.2	1.2	1.2	
EXECUTE	0.8 + time for instruction to be executed			
EXTENDED OPERATION	4.8	5.8	5.05	5
HALVE	1.0	1.0	1.0	
HEX SHIFT LEFT	1.8	3.0	depends on shift count	
HEX SHIFT RIGHT	1.8	3.0	depends on shift count	
INCLUSIVE OR	0.6	0.6	0.6	
INCLUSIVE OR IMMEDIATE	1.2	1.2	1.2	
INCREMENT	0.6	0.6	0.6	1
INCREMENT AND SKIP IF ZERO	1.4	1.6	1.5	4
EXTENDED INCREMENT AND SKIP IF ZERO	2.4	2.6	2.45	4

APPENDIX F (Continued)

INSTRUCTION EXECUTION TIMES

STANDARD INSTRUCTION SET	MINIMUM	MAXIMUM	TYPICAL	NOTES
JUMP	0.6	0.8	0.65	4
EXTENDED JUMP	1.6	1.6	1.6	4
JUMP TO SUBROUTINE	0.6	0.8	0.65	4
EXTENDED JUMP TO SUBROUTINE	1.6	1.6	1.6	4
LOAD ACCUMULATOR	0.8	1.4	1.0	4
EXTENDED LOAD ACCUMULATOR	1.8	2.4	1.95	4
EXTENDED LOAD EFFECTIVE ADDRESS	1.6	1.6	1.6	4
LOAD BYTE	1.4	1.8	1.5	
LOAD MAP	2.4+0.6N	2.4+0.6N	2.3+0.6N	2
LOCATE AND RESET LEAD BIT	1.2	7.2	1.2+0.4N	3
LOCATE LEAD BIT	1.0	7.0	1.0+0.4N	3
LOGICAL SHIFT	1.0	3.8	depends on shift count	
MODIFY STACK POINTER	2.2	2.4	2.25	12
MOVE	0.6	0.6	0.6	1
NEGATE	0.6	0.6	0.6	1
POP BLOCK	4.0	4.8	4.2	7
POP MULTIPLE ACCUMULATORS	2.2+0.4N	3.0+0.4N	2.4+0.4N	7,8
POP PC AND JUMP	2.4	3.2	2.6	7
PUSH JUMP	2.2	3.6	2.7	6,13
PUSH MULTIPLE ACCUMULATORS	2.2+0.4N	3.0+0.4N	2.4+0.4N	5,8
PUSH RETURN ADDRESS	2.6	3.2	2.8	5
RESTORE	6.0	8.0	6.85	
RETURN	4.4	5.0	4.55	7
SAVE	3.8	5.2	4.08	5
SET BIT TO ONE	2.4	2.8	2.45	6
SET BIT TO ZERO	2.4	2.8	2.45	6
SIGN EXTEND AND DIVIDE	2.2	9.8	9.5	
SIGNED DIVIDE	2.2	10.2	9.6	
SIGNED MULTIPLY	7.2	7.2	7.2	
SKIP IF ACS > ACD	1.0	1.0	1.0	
SKIP IF ACS $\geq$ ACD	1.0	1.0	1.0	
SKIP ON NON-ZERO BIT	2.2	2.6	2.3	6,10
SKIP ON ZERO BIT	2.2	2.6	2.3	6,10
SKIP ON ZERO BIT AND SET TO ONE	2.6	2.8	2.8	6
STORE ACCUMULATOR	0.8	1.4	1.0	4
EXTENDED STORE ACCUMULATOR	1.8	2.4	1.95	4
STORE BYTE	1.8	2.0	1.85	
SUBTRACT	0.6	0.6	0.6	1
SUBTRACT IMMEDIATE	0.6	0.6	0.6	
SYSTEM CALL	4.2	5.0	4.45	5,6
UNSIGNED DIVIDE	1.6	8.2	8.2	
UNSIGNED MULTIPLY	7.2	7.2	7.2	

APPENDIX F (Continued)

INSTRUCTION EXECUTION TIMES

I/O INSTRUCTION SET	MINIMUM	MAXIMUM	TYPICAL	NOTES
DATA INPUT	2.2	2.2	2.2	9
DATA OUTPUT	2.6	2.6	2.6	9
I/O SKIP	0.8	0.8	0.8	10
NO I/O TRANSFER	1.2	1.2	1.2	9
VECTOR ON INTERRUPTING DEVICE CODE				
MODE A	2.6	2.8	2.65	
MODE B	8.6	9.6	8.85	5
MODE C	10.2	12.2	10.75	5
MODE D	15.0	18.0	16.5	5
MODE E	16.6	20.2	18.05	5
PROGRAM INTERRUPT CYCLE	1.4	1.8	1.5	4
DATA CHANNEL INPUT	0.8	0.8	0.8	
DATA CHANNEL OUTPUT	1.4	1.6	1.6	
DATA CHANNEL LATENCY	0.8	5.6	1.5	

PROGRAM INTERRUPT LATENCY is the sum of the longest time that the program runs with the interrupt system disabled plus the time for the program interrupt cycle. The longest non-interruptable instruction is a Mode E VECTOR with a maximum time of 20.2 micro-seconds. Because this VECTOR also enables the interrupt system, an interrupt will not be honored until after the next instruction, unless the next instruction is interruptable (e.g., BLOCK ADD AND MOVE). Therefore, the time for the next longest instruction must be added. The next longest instruction is COUNT BITS WITH A worst case time of 10.6 microseconds. To this must be added the time for the program interrupt cycle of 1.8 micro-seconds. This yields an absolute worst case program interrupt latency of 32.6 microseconds. The interrupt latency for a specific application can be computed using the above method.

APPENDIX F (Continued)
INSTRUCTION EXECUTION TIMES

NOTES:		MINIMUM	MAXIMUM	TYPICAL
1.	If skip occurs, add:	0.6	0.6	0.6
2.	N is number of words moved.			
	For each indirect reference in AC3, add:	0.8	0.8	0.8
	For each indirect reference in AC2, add:	0.6	0.8	0.65
	If N is less than 1, then time is:	1.2	1.2	1.2
3.	N is the count added to ACD.			
	For LOCATE AND RESET LEAD BIT, if the count is 16, the time is:	7.4	7.4	7.4
	For LOCATE LEAD BIT, if the count is 16, the time is:	7.2	7.2	7.2
4.	For each indirect reference, add:	0.6	0.8	0.65
	For each indirect auto-index reference, add:	1.0	1.6	1.15
5.	If stack overflows, add:	3.2	3.8	3.45
	In addition, see note 6.			
6.	For each indirect reference, add:	0.8	0.8	0.8
7.	If stack underflows and underflow protection is disabled, add:	0.4	0.8	0.7
	If stack underflows and underflow protection is enabled, add:	3.8	5.0	4.4
	In addition, see note 6.			
8.	N is number of words pushed or popped.			
9.	S, C, and P functions require no extra time.			
10.	If skip occurs, add:	0.4	0.4	0.4
11.	For each indirect reference in the table address add:	0.6	0.6	0.6
12.	If stack overflows add:	4.2	4.8	4.65
13.	If stack overflows add:	4.0	4.6	4.25

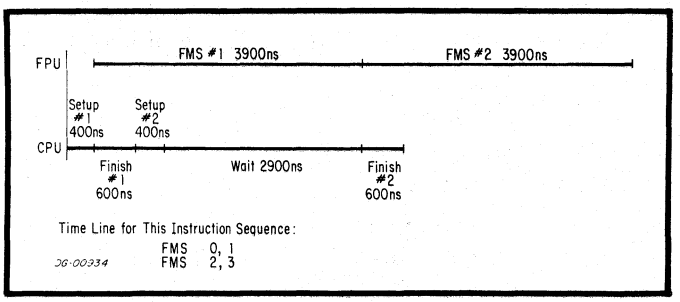
APPENDIX F (Continued)
INSTRUCTION EXECUTION TIMES

FLOATING POINT INSTRUCTION EXECUTION TIMES

Because the CPU and the floating point feature operate in parallel, there are two distinct times to consider when dealing with the execution time of a floating point instruction. These are "FPU time" and "CPU time".

FPU time is the amount of time taken in the floating point unit actually performing the calculation.

CPU time is that amount of time that the CPU devotes to a floating point instruction. This time is divided into three parts: setup time, wait time, and finish time. Setup time is the time devoted to decoding the instruction and computing the effective address if required. Wait time is the time spent by the CPU waiting for the FPU to finish a previous operation and become idle. Finish time is the time devoted to transferring to the FPU all required operands and initiating the floating point operation. The following example illustrates these times.



Wait time is given by the following equation:

WAIT= FPU time for previous instruction - (finish time for previous instruction + total execution time for non-floating point instructions between the floating point instructions + setup time for this floating point instruction).

If WAIT is less than 0, then a value of 0 should be used for WAIT.

APPENDIX F (Continued)
INSTRUCTION EXECUTION TIMES

INSTRUCTION	CPU SETUP	CPU FINISH	FPU	REMARKS
ADD SINGLE (FPAC) ADD DOUBLE (FPAC) SUBTRACT SINGLE (FPAC) SUBTRACT DOUBLE (FPAC)	0.4	0.6	1.5 2.3 2.4 1.9	Exponent over- or underflow Mantissa overflow Normalization needed Normalization not needed
ADD SINGLE (MEMORY) SUBTRACT SINGLE (MEMORY)	1.2 (Note 1)	0.8	2.2 3.0 3.1 2.6	Exponent over- or underflow Mantissa overflow Normalization needed Normalization not needed
ADD DOUBLE (MEMORY) SUBTRACT DOUBLE (MEMORY)	1.2 (Note 1)	1.4	2.8 3.6 3.7 3.2	Exponent over- or underflow Mantissa overflow Normalization needed Normalization not needed
MULTIPLY SINGLE (FPAC)	0.4	0.6	3.9	
MULTIPLY DOUBLE (FPAC)	0.4	0.6	7.1	
MULTIPLY SINGLE (MEMORY)	1.2	0.8	4.6	Note 1
MULTIPLY DOUBLE (MEMORY)	1.2	1.4	8.4	Note 1
DIVIDE SINGLE (FPAC)	0.4	0.6	4.2 5.1	Divisor mantissa $>$ dividend mantissa Divisor mantissa $\leq$ dividend mantissa
DIVIDE DOUBLE (FPAC)	0.4	0.6	7.4 8.3	Divisor mantissa $>$ dividend mantissa Divisor mantissa $\leq$ dividend mantissa
DIVIDE SINGLE (MEMORY)	1.2 (Note 1)	0.8	4.9 5.8	Divisor mantissa $>$ dividend mantissa Divisor mantissa $\leq$ dividend mantissa
DIVIDE DOUBLE (MEMORY)	1.2 (Note 1)	1.4	8.7 9.6	Divisor mantissa $>$ dividend mantissa Divisor mantissa $\leq$ dividend mantissa
LOAD SINGLE	1.2	0.8	1.6	Note 1
LOAD DOUBLE	1.2	1.4	2.2	Note 1
STORE SINGLE	1.2	0.8	0.5	Notes 1, 2
STORE DOUBLE	1.2	1.2	0.5	Notes 1, 2
FLOAT FROM AC	0.4	0.6	1.9 2.3	Integer positive Integer negative
FLOAT FROM MEMORY	1.2 (Note 1)	0.8	2.3 2.7	Integer positive Integer negative
FIX TO AC	0.4 (Note 2)	0.6	2.1 2.5	Integer positive Integer negative
FIX TO MEMORY	1.2 (Notes 1, 2)	0.8	2.3 2.7	Integer positive Integer negative

APPENDIX F (Continued)
INSTRUCTION EXECUTION TIMES

	CPU		FPU	REMARKS
	SETUP	FINISH		
NEGATE	0.4	0.6	1.3	
ABSOLUTE VALUE	0.4	0.6	1.3	
READ HIGH WORD	0.4	0.6	0.4	Note 2
SCALE	0.6	0.6	1.7	
LOAD EXPONENT	0.6	0.6	1.6	
HALVE	0.8	0.6	1.8	
MOVE	0.4	0.6	1.0	
NORMALIZE	0.4	0.6	1.4	
COMPARE	0.4	0.6	0.9	
LOAD STATUS	1.6	0.8	0.7	Notes 1, 2
STORE STATUS	1.6	0.8	0.5	Notes 1, 2
PUSH FLOATING POINT STATE	1.4	7.0	7.0	
POP FLOATING POINT STATE	1.4	8.4	8.4	
TRAP ENABLE TRAP DISABLE CLEAR ERRORS	1.0	0.6	0.4	
SKIP TESTS	0.4	0.6	0.4	Note 3

NOTES:
1. For setup time, add 0.4 for first indirect reference and 0.6 for each subsequent indirect reference. For finish time, add 0.2 for each indirect reference except for store instructions, add 0.0 for each indirect reference.
2. FPU time can begin concurrently with the beginning of setup time, if the FPU is idle. Otherwise, FPU time begins as soon as the FPU finishes the previous instruction. Finish time cannot commence until the FPU has completed this instruction.
3. If skip occurs, add 0.2 to finish time.

APPENDIX G
USE OF THE VECTOR INSTRUCTION

The VECTOR ON INTERRUPTING DEVICE CODE instruction is an extremely powerful instruction. Because of the impact of interrupt latency on overall system performance, and the impact of the VECTOR instruction on interrupt latency, this instruction should be well understood before it is used.

The VECTOR instruction can operate in any one of five modes. These modes are called mode A, mode B, mode C, mode D, and mode E. In general, as one goes through the modes, from A to E, the instruction performs more work, giving the user more power, but also takes more time to execute.

For all modes, the VECTOR instruction uses bits 1-15 of the second word to address the vector table. An INTERRUPT ACKNOWLEDGE instruction is performed and the device code received is added to the address of the vector table and the word at that location is fetched. At this point, the mode selection process begins.

Which mode is used for execution is a function of the direct bit in the vector table entry, the stack change bit in the second word of the VECTOR instruction and the push bit in the first word of the DCT. The table below gives the relationship.

DIRECT	STACK CHANGE	PUSH	MODE
0	X	X	A
1	0	0	B
1	0	1	C
1	1	0	D
1	1	1	E

Note: X = Don't care

For mode A, the state of the stack change and push bits don't matter because they are never checked.

The uses of the five modes are described below.

Mode A is used when no time can be wasted in getting to the interrupt handler for a device. A device requiring mode A service would typically be a non-buffered device with a very small latency time. Alternatively, a real time process that must receive control immediately after an event could be serviced using mode A. The programmer pays for the speed realized through mode A by giving up the state saving and priority masking features of the other modes.

Modes B, C, D, and E are used to implement a priority interrupt structure. They all build a new priority mask and save the old priority mask before issuing a MASK OUT instruction that enables the interrupt system. These modes differ in the amount of time and work that they devote to saving the state of the machine.

In a priority system, there are typically two types of processes: those that operate at "base" level, and those that do not. Base level is defined as operating with all levels of interrupt enabled and no interrupt processing in progress. Non-base level is defined as operating with some interrupt processing in progress. In general, those processes that operate at base level are user problem programs. Those processes that operate at non-base level are the various interrupt handlers in the system.

One of the first things that the supervisor program should do when it receives an interrupt while a process is operating at base level is to change the stack environment. Two reasons lead to this conclusion. The supervisor has no control over whether or not the user has defined a stack by placing meaningful information in the stack control words. Additionally, even if the user has initialized a stack, the supervisor has no control over the size of the stack. If the user has defined a stack, but

APPENDIX G (Continued)

USE OF THE VECTOR INSTRUCTION

is very close to his stack limit, it would not be acceptable for a supervisor interrupt routine to fill the user's stack to overflowing. By using either mode D or E, the VECTOR instruction will change the stack environment and initialize a stack over which the supervisor has full control. At the same time, the VECTOR instruction will save the stack environment of the user so that it may be restored before control is returned to the user.

If an interrupt handler is already processing when another interrupt is received, then the stack environment has already been changed by the interrupt that occurred at base level and should not be changed again. For interrupts that occur at non-base level, modes B and C of the VECTOR instruction can be used.

The difference between modes D and E is the same as the difference between modes B and C: Modes B and D do not push a return block onto the stack. While this saves a little bit of time over modes C and E, it makes returning control to the interrupted program somewhat more complicated.

All modes of the VECTOR instruction can be combined in one vector table. Devices that require mode A service will have bit 0 set to 0 in their vector table entry. The other devices will have bit 0 set to 1 in their vector table entries, and control their modes of service by the setting of the push bit in their DCT's.

The following example illustrates the use of the VECTOR instruction. This example assumes a system with only three peripherals: An event counter that requires mode A service; a slow speed input device (TTY input); and a slow speed output device (TTY output). The output device is of lower priority than the input device. The execution times of the instructions illustrated here can be found in Appendix F.

APPENDIX G (Continued)
USE OF THE VECTOR INSTRUCTION

```
        .LOC    0           ;START ASSEMBLY AT 0
INTR:   0                   ;INTERRUPT RETURN
INTE:   PI                  ;ADDRESS OF PROGRAM INTERRUPT ROUTINE
SC:     SCH                 ;ADDRESS OF SCL HANDLER
PF:     PFH                 ;ADDRESS OF PF HANDLER
VSP:    VS                  ;VECTOR STACK POINTER
CURMK:  0                   ;CURRENT MASK
VSL:    VL                  ;VECTOR STACK LIMIT
VSF:    VF                  ;VECTOR STACK FAULT HANDLER

        .LOC    50          ;NEXT LOCATION IS 50 OCTAL
LEVEL:  -1                  ;INTERRUPT LEVEL COUNT
        .LOC    1000        ;NEXT LOCATION IS 1000 OCTAL
PI:     ISZ     LEVEL       ;BASE LEVEL?
        JMP     .+3         ;NO
        VCT     @VTAB       ;BASE LEVEL VECTOR--
                            ; @ SETS STACK CHANGE BIT TO 1
        VCT     VTAB        ;NON-BASE LEVEL VECTOR--
                            ; STACK CHANGE BIT IS 0

DISMIS: POP     3,3         ;POP OLD MASK INTO AC3
        DOBC    3,CPU       ;MASK OUT THAT DISABLES INTERRUPTS
        STA     3,CURMK     ;STORE MASK INTO CURRENT MASK
        LDA     3,LEVEL     ;PICK UP LEVEL
        SBI     1,3         ;SUBTRACT 1
        STA     3,LEVEL     ;STORE IT BACK
        COM#    3,3,SZR     ;BASE LEVEL?
        JMP     .+3         ;NO--JUST RETURN
        INTEN               ;YES
        RSTR                ;RESTORE
        INTEN               ;INTERRUPT ENABLE
        POPB                ;RETURN


VTAB:   @SPUR               ;SPUR IS ADDRESS OF DCT FOR SPURIOUS INTERRUPT ROUTINE
        @SPUR
        @SPUR
        @SPUR
        EVEN                ;EVEN IS ADDRESS OF EVENT INTERRUPT HANDLER
        @SPUR
        @SPUR
        @SPUR
        @TTIN               ;ADDRESS OF DCT FOR TTY INPUT
        @TTOUT              ;ADDRESS OF DCT FOR TTY OUTPUT
        @SPUR               ;REST OF TABLE IS FILLED WITH SPUR
        ...
        ...
        ...
SPUR:   @SPURH              ;PUSH BIT = 1--SPURH = ADDRESS OF SPURIOUS INTERRUPT HANDLER
        0                   ;DO NOT CHANGE CURRENT MASK
TTIN:   @TTIH               ;TTIH = ADDRESS OF TTI INTERRUPT HANDLER
        3                   ;MASK OUT LEVEL 14 AND 15
TTOUT:  @TTOH               ;TTOH = ADDRESS OF TTO INTERRUPT HANDLER
        1                   ;MASK OUT LEVEL 15
        ...
        ...
        ...
EVEN:   ...                 ;DO PROCESSING ASSOCIATED WITH EVENT COUNTER
        ...
        JMP     @0          ;RETURN TO ADDRESS IN LOC 0
TTIH:   ...                 ;DO PROCESSING ASSOCIATED WITH TTY INPUT
        ...
        ...
        JMP     DISMIS      ;GO TO DISMISS ROUTINE
TTOH:   ...                 ;DO PROCESSING ASSOCIATED WITH TTY OUTPUT
        ...
        JMP     DISMIS      GO TO DISMISS ROUTINE
```

APPENDIX H

INSTRUCTION USE EXAMPLES

On the following pages are examples of how the instruction set of the ECLIPSE computer may be used to perform some common functions.

1. Clear an AC and the carry bit.

```
SUBO    AC,AC
```

2. Clear an AC and preserve the carry bit.

```
SUBC    AC,AC
```

3. Generate the indicated constants.

```
SUBZL   AC,AC       ;GENERATE +1
ADC     AC,AC       ;GENERATE -1
ADCZL   AC,AC       ;GENERATE -2
```

4. Let ACX be any accumulator whose contents are zero. Generate the indicated constants in ACX.

```
INCZL   ACX,ACX     ;GENERATE +2
INCOL   ACX,ACX     ;GENERATE +3
INCS    ACX,ACX     ;GENERATE +400(8)
```

5. Check if both bytes in an accumulator are equal.

```
MOVS    ACS,ACD
SUB     ACS,ACD,SZR
JMP     ---         ;NOT EQUAL
---     ---         ;EQUAL
```

6. Check if two accumulators are both zero.

```
MOV     ACS,ACS,SNR
SUB     ACS,ACD,SZR
JMP     ---         ;NOT BOTH ZERO
---     ---         ;BOTH ZERO
```

7. Check an ASCII character to make sure it is a decimal digit. The character is in ACS and is not destroyed by the test. Accumulators ACX and ACY are destroyed.

```
LDA     ACX,C60     ;ACX=ASCII ZERO
LDA     ACY,C71     ;ACY=ASCII NINE
ADCZ#   ACY,ACS,SNC ;SKIPS IF (ACS) > 9
ADCZ#   ACS,ACX,SZC ;SKIPS IF (ACS) ≥ 0
JMP     ---         ;NOT DIGIT
---     ---         ;DIGIT

C60:    60          ;ASCII ZERO
C71:    71          ;ASCII NINE
```

8. Test an accumulator for zero.

```
MOV     AC,AC,SZR
JMP     ---         ;NOT ZERO
---     ---         ;ZERO
```

APPENDIX H (Continued)

INSTRUCTION USE EXAMPLES

9. Test an accumulator for -1.

```
COM#    AC,AC,SZR
JMP     ---             ;NOT -1
---     ---             ;-1
```

10. Test an accumulator for 2 or greater.

```
MOVZR#  AC,AC,SNR
JMP     ---             ;LESS THAN 2
---     ---             ;2 OR GREATER
```

11. Assume it is known that AC contains 0, 1, 2, or 3. Find out which one.

```
MOVZR#  AC,AC,SEZ
JMP     THREE           ;WAS 3
MOV     AC,AC,SNR
JMP     ZERO            ;WAS 0
MOVZR#  AC,AC,SZR
JMP     TWO             ;WAS 2
---     ---             ;WAS 1
```

12. Multiply an AC by the indicated value.

```
MOV     ACX,ACX         ;MULTIPLY BY 1

MOVZL   ACX,ACX         ;MULTIPLY BY 2

MOVZL   ACX,ACY         ;MULTIPLY BY 3
ADD     ACY,ACX

ADDZL   ACX,ACX         ;MULTIPLY BY 4

MOV     ACX,ACY
ADDZL   ACX,ACX         ;MULTIPLY BY 5
ADD     ACY,ACX

MOVZL   ACX,ACY         ;MULTIPLY BY 6
ADDZL   ACY,ACX

ADDZL   ACX,ACX         ;MULTIPLY BY 8
MOVZL   ACX,ACX
```

Multiplication by other factors of 2 can be achieved with the LOGICAL SHIFT instruction. Multiplication by factors of 16 can be accomplished with the HEX SHIFT LEFT instruction.

13. Perform the following unsigned integer comparisons.

```
SUB#    ACS,ACD,SZR     ;SKIP IF CONTENTS OF ACS = CONTENTS OF ACD
SUB#    ACS,ACD,SNR     ;SKIP IF CONTENTS OF ACD ≠ CONTENTS OF ACD
ADCZ#   ACS,ACD,SNC     ;SKIP IF CONTENTS OF ACS < CONTENTS OF ACD
SUBZ#   ACS,ACD,SNC     ;SKIP IF CONTENTS OF ACS ≤ CONTENTS OF ACD
SUBZ#   ACS,ACD,SZC     ;SKIP IF CONTENTS OF ACS > CONTENTS OF ACD
ADCZ#   ACS,ACD,SZC     ;SKIP IF CONTENTS OF ACS ≥ CONTENTS OF ACD
```

APPENDIX I

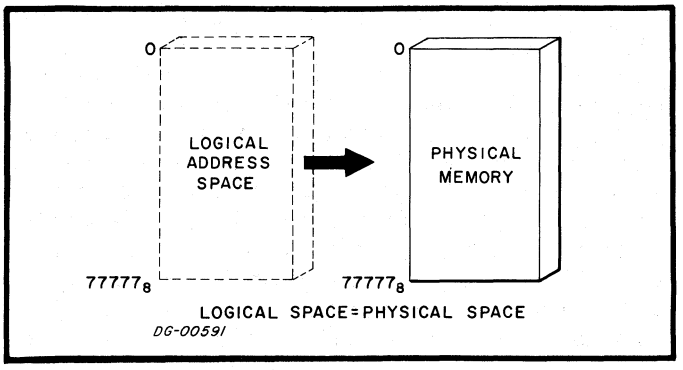
S/200 AND C/300 MAP

MEMORY ALLOCATION AND PROTECTION FEATURE

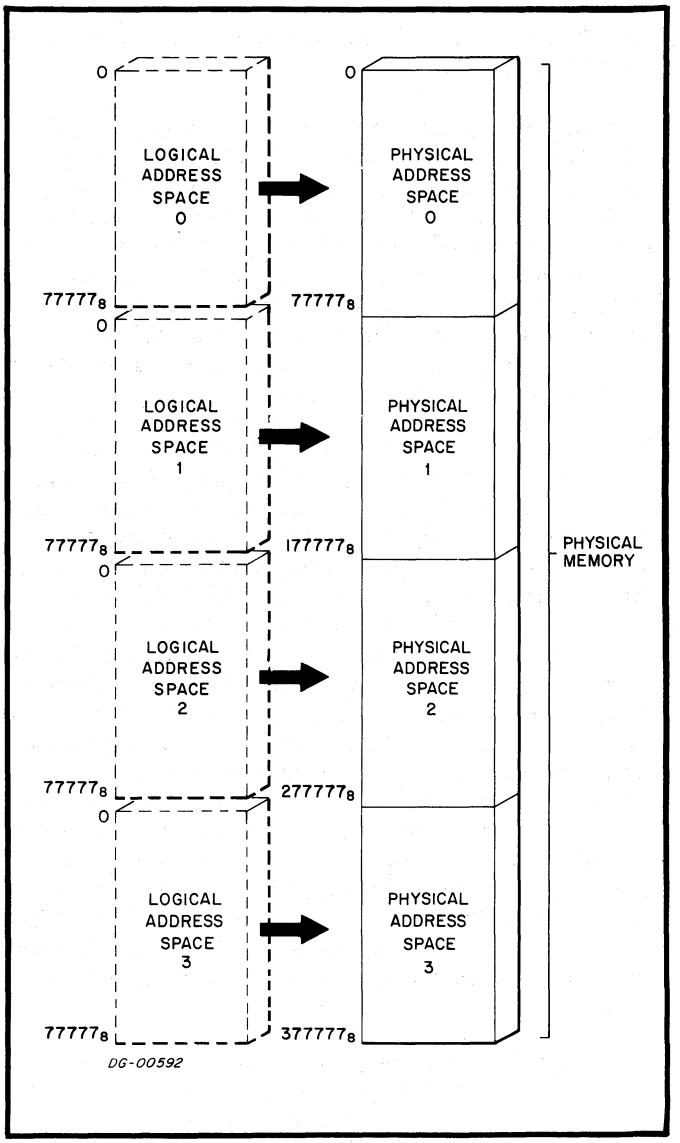
Introduction to Address Translation

The Memory Allocation and Protection (MAP) feature provides a way to efficiently allocate to users the memory and peripheral resources available in an ECLIPSE line system. In addition, the resources, once allocated, can be protected from unauthorized access by another user. The MAP feature also allows the size of physical memory to be increased from 64K bytes to 256K bytes.

The process used by the MAP feature to assist in the allocation of memory is called "logical to physical address translation". As stated before, the "address space" available to a user consists of the 32,768 2-byte memory locations from 0 to 77777_8. This user address space is called the "logical" address space. The physical main memory available to the CPU is called the "physical" address space. If the MAP feature is not installed, the maximum size of the physical address space is limited to 64K bytes and the logical address space is equal to the physical address space. In other words, physical location 0 is always used to hold logical location 0, physical location 1 is always used to hold logical location 1, and so on.



With the MAP feature installed the maximum size of the physical address space is increased to 256K bytes. The maximum size of the logical address space is not increased, however, and is still 64K bytes. This means that the physical address space is now big enough to hold four mutually exclusive logical address spaces at the same time.



In the above illustration, physical locations $0-77777_8$ are used to hold an entire logical address space. Physical locations $100000_8-177777_8$ are used to hold a different logical address space. Physical locations $200000_8-277777_8$ are used to hold a third logical address space and physical locations $300000_8-377777_8$ are used to hold a fourth logical address space. It can be seen from this illustration that while there is only one physical location 0, there are four logical locations with the address 0. Physical locations 0, 100000_8, 200000_8, and 300000_8 are each used to hold a logical location 0 for a different logical address space. The physical location corresponding to a given log-

ical location in any of the four logical address spaces could be found by performing the following computation:
((logical space#)*(100000_8))+(logical address) = physical address

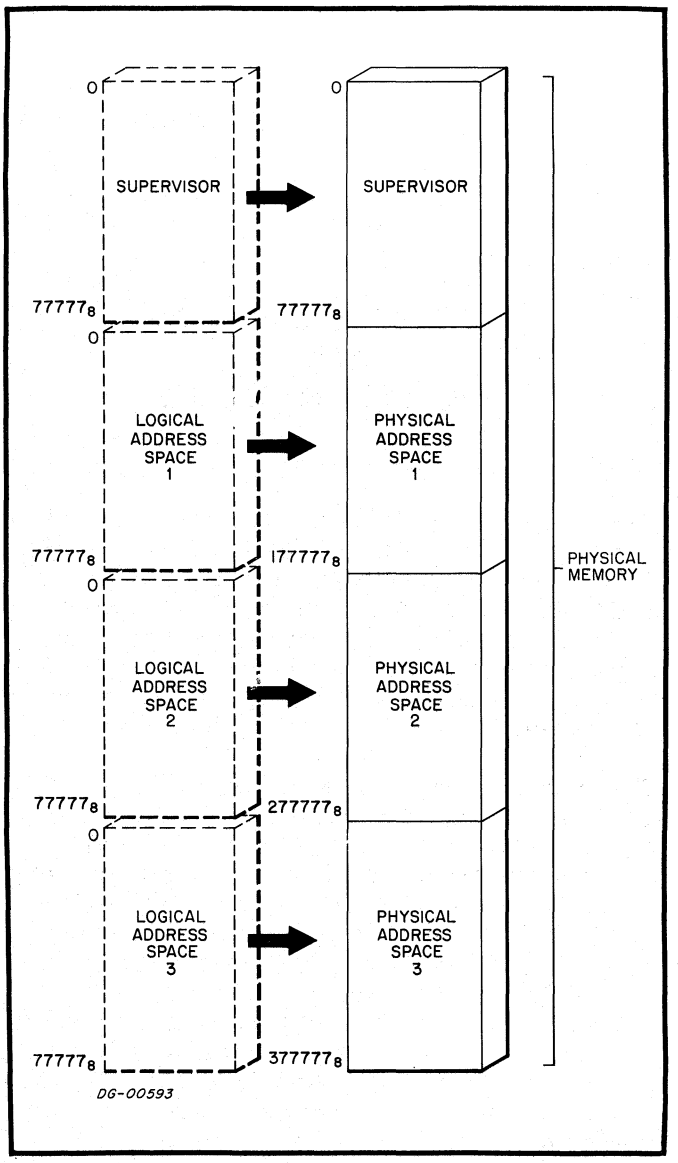
For example take logical address 501_8 in logical space 2:
((2)*(100000_8))+(501_8) = 200501_8

In other words, physical location 200501_8 is used to hold the word at logical address 501_8 in logical address space 2.

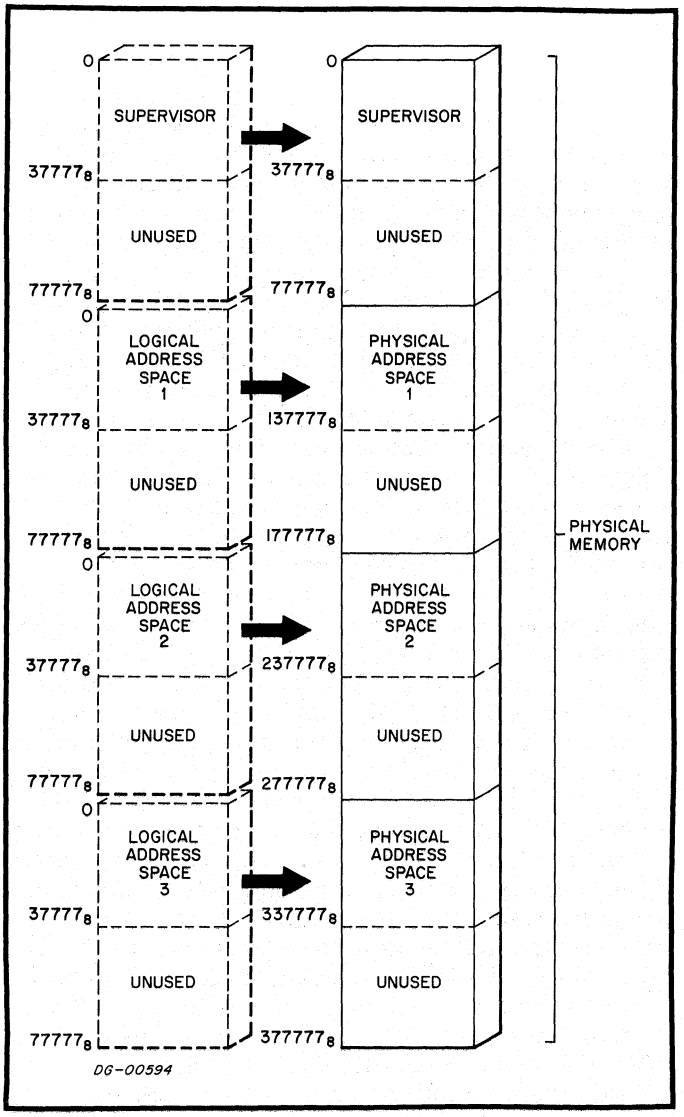
The same address, but in logical space 3 corresponds to physical address 300501_8. If every memory reference coming from the CPU were a logical address, and if it were translated using the above computation before the actual memory reference were made, this setup would allow four user programs to run in the enlarged physical space. In addition, each user would perform as if it were the only user of an ECLIPSE line computer without the MAP feature.

There are two problems with this scheme of inserting small logical address spaces into a relatively large physical memory space. First of all, a supervisory program is needed to monitor the actions of the user programs. This supervisor is responsible for allocating a block of physical memory to a user, for loading the user's program into the allocated physical memory and for determining the order in which the loaded user programs will receive the instruction execution services of the CPU. These functions could possibly be performed by the MAP feature itself, but all generality would be lost. In any case, to implement these functions within the MAP feature would be quite costly. This supervisor program must, obviously, occupy a region of physical memory that would otherwise be allocated to a user. This means that the number of users that could be serviced at one time would be reduced from four to three.

NOTE In general, the supervisor could occupy any of the four regions, but because it is simpler to implement a supervisor that resides in the lowest region of physical memory, it will be assumed that the supervisor occupies the region of physical memory allocated to logical block 0. In other words, the supervisor operates with its logical address space equal to its physical address space and no address translation is performed.



Secondly, this scheme would lead to inefficient use of physical memory. In all probability, the user programs being serviced would not need all of the 64K bytes of physical memory allocated to them. Certainly, the supervisor would not need all of the 64K bytes allocated to it. Unfortunately, any unused physical memory would be wasted. Suppose that the supervisor and each of the three users were using only 32K of their 64K allocated bytes. This would leave 128K bytes unused. This would be enough physical memory to hold two entire logical address spaces. Alternatively, this 128K of physical memory could be used to service four additional users if they needed only 32K bytes each.



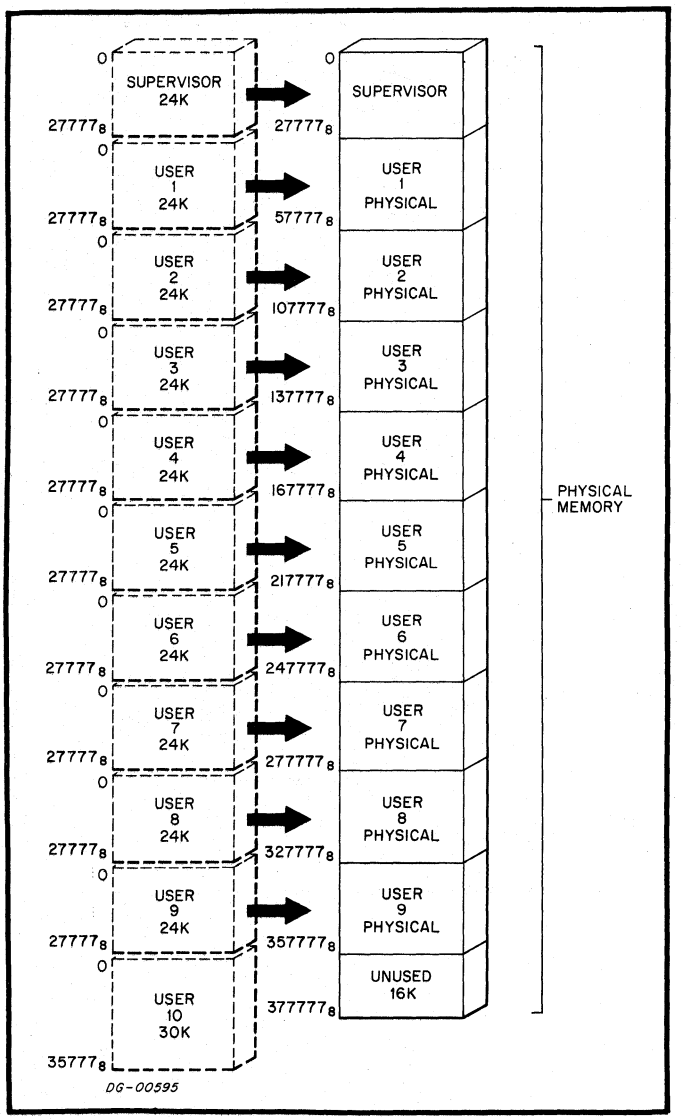
Address Translation on the ECLIPSE Computers

A supervisor program is still needed when processing with the MAP feature, but the amount of wasted physical memory is greatly reduced. The MAP feature allows more efficient allocation of physical memory by allowing physical memory to be allocated in blocks of 2K bytes, instead of the blocks of 64K bytes used in the above example. In addition, the MAP feature allows a different logical to physical address computation to be specified for each 2K byte block of logical memory. In addition, allocated blocks of physical memory do not have to be contiguous.

The allocation of physical memory in blocks of 2K bytes reduces waste of physical memory in two ways. It means that the amount of physical memory allocated to a user need be no greater than the amount of physical memory the user requires, rounded up to the next 2K bytes. It also means that there are many more blocks of physical memory available for allocation. In the above example, there were four 64K byte blocks available for allocation. By allocating physical memory in blocks of 2K bytes, the physical address space of 256K bytes is broken into 128 different allocatable blocks. A new block begins every 2K. The only restriction on the allocation of physical memory is that the first physical address in a block of allocated physical memory must be a multiple of 2K. This means that a block can start at physical location 4000_8 or 10000_8, but not at 12000_8.

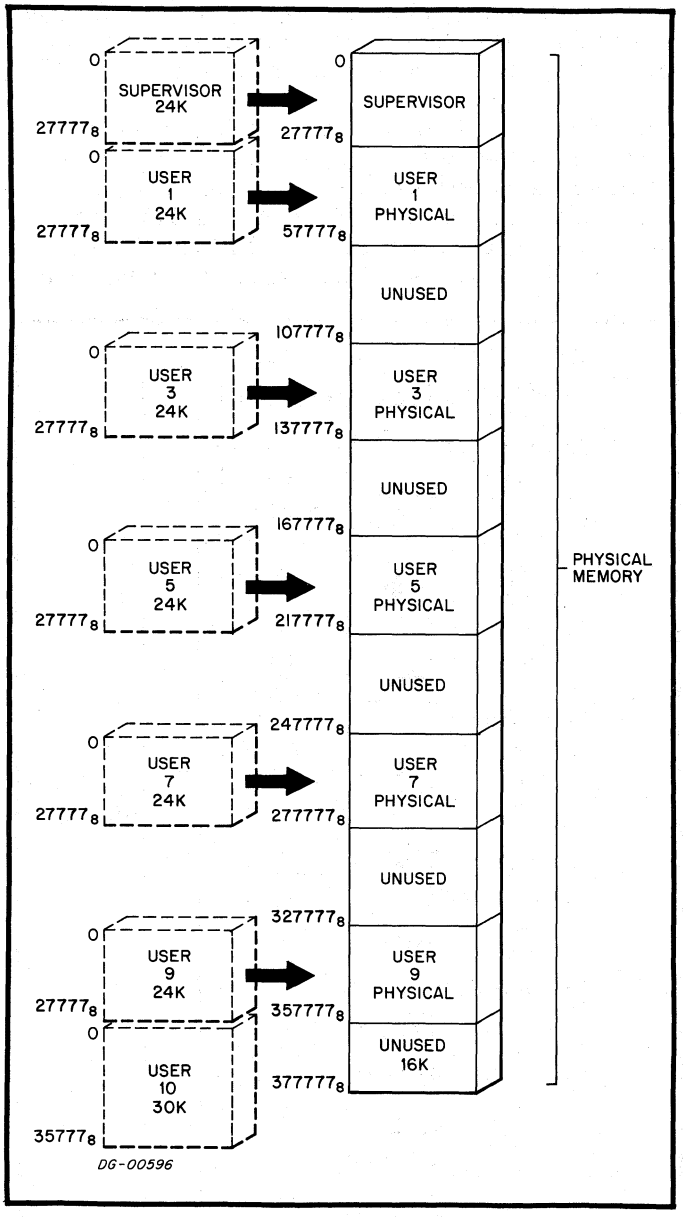
If only one logical to physical address computation could be specified for each user, all the blocks of physical memory allocated to that user would have to be contiguous. The first 2K bytes of that user's logical address space would reside in the lowest addressable 2K block allocated to him. The next 2K bytes of logical addresses would reside in the next 2K of physical memory, and so on. While physical memory waste would be less than for the case of 64K byte blocks, a significant amount of physical memory waste could still occur.

Consider the case of a system with a supervisor that runs in 24K bytes. This leaves 232K bytes available for allocation to users. Assume that there are nine users, each requiring 24K bytes, and a tenth user that requires 30K bytes. The supervisor allocates the necessary physical mem-



ory to the first nine users, using up 216K bytes of the 232K bytes available. Obviously, the tenth user cannot be serviced because there are only 16K bytes unused.

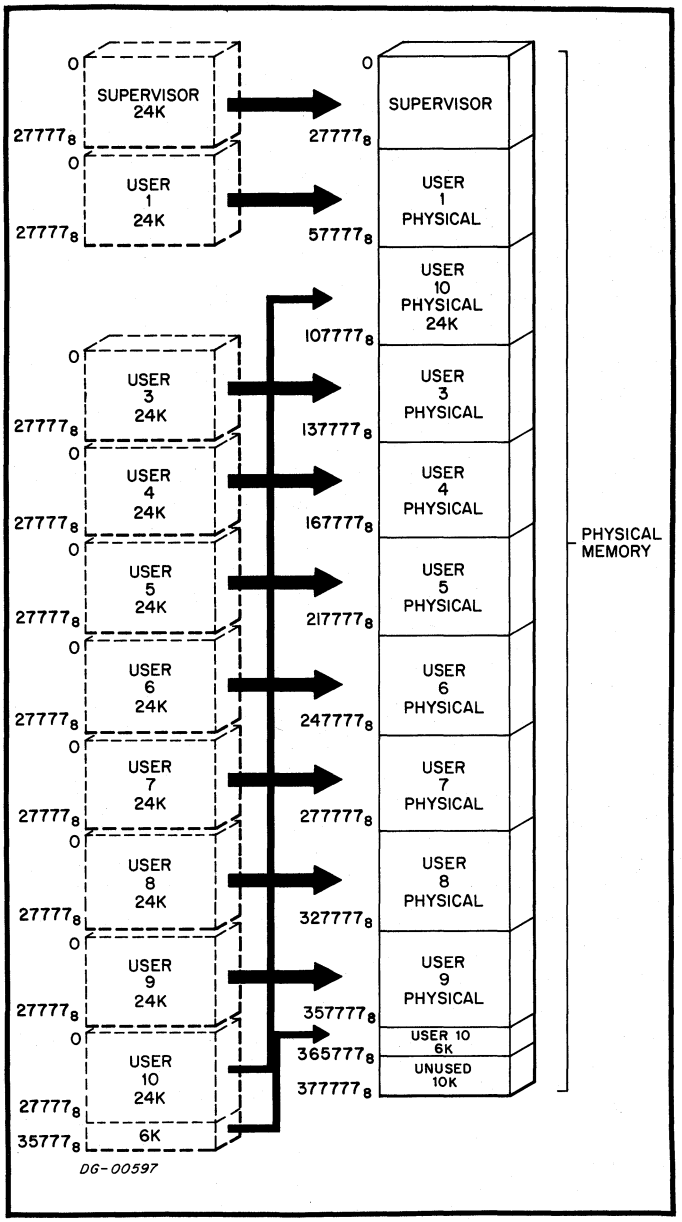
Now, assume that user 2 finishes his job. The supervisor knows that there are now 30K bytes available, but cannot service user 10 because the 30K bytes are not contiguous. If user 4 finishes his job next and then user 6 and finally user 8, the supervisor knows that 112K bytes are available, but still user 10 cannot be serviced because there is not a contiguous block of 30K bytes. User 10 must wait until one of users 1, 3, 5, 7, or 9 finishes.



Because the MAP feature allows a logical to physical address computation to be specified for each block of logical memory, this situation cannot occur. When the supervisor allocates a block of physical memory to hold a 2K byte block of a logical address space it gives to the MAP feature the number of the logical block and the number of the corresponding physical block. In reality, the number of the logical block is the high-order five bits of the first address in that logical block. The number of the corresponding physical block is the high-order seven bits of the first address in that physical block. Given this information, all the MAP feature has to do to translate a logical address to a physical address is to make the correspondence between the logical block number and the physical block number and then append the low-order ten bits of the logical address to the physical block number. This is the process that translates a 15-bit logical address to a 17-bit physical address. This procedure is called "mapping" an address. The logical to physical address computation for a specific block is called the "map" for that block. The set of address translation computations that completely defines where a user's logical space resides in physical memory is called the "user map" for that user. The number of the user does not enter into this procedure because the MAP feature only translates addresses for one user at any one time. The supervisor controls which user will receive the instruction execution services of the CPU by directing the MAP feature to translate addresses using only the logical to physical address computations for a specific user. When the supervisor decides that it is time for a different user to receive the services of the CPU, the supervisor tells the MAP feature to invalidate the current user map. The supervisor then gives the MAP feature a new user map and directs the MAP feature to translate addresses using the new set of computations.

The waste of physical memory outlined above cannot occur because with a translation computation for each block of logical memory, the blocks do not have to be contiguous. In fact, they do not have to be in any order at all. Because each block of a user's logical address space is individually tied to the corresponding block of physical memory, the blocks of physical memory can be anywhere in the physical address space.

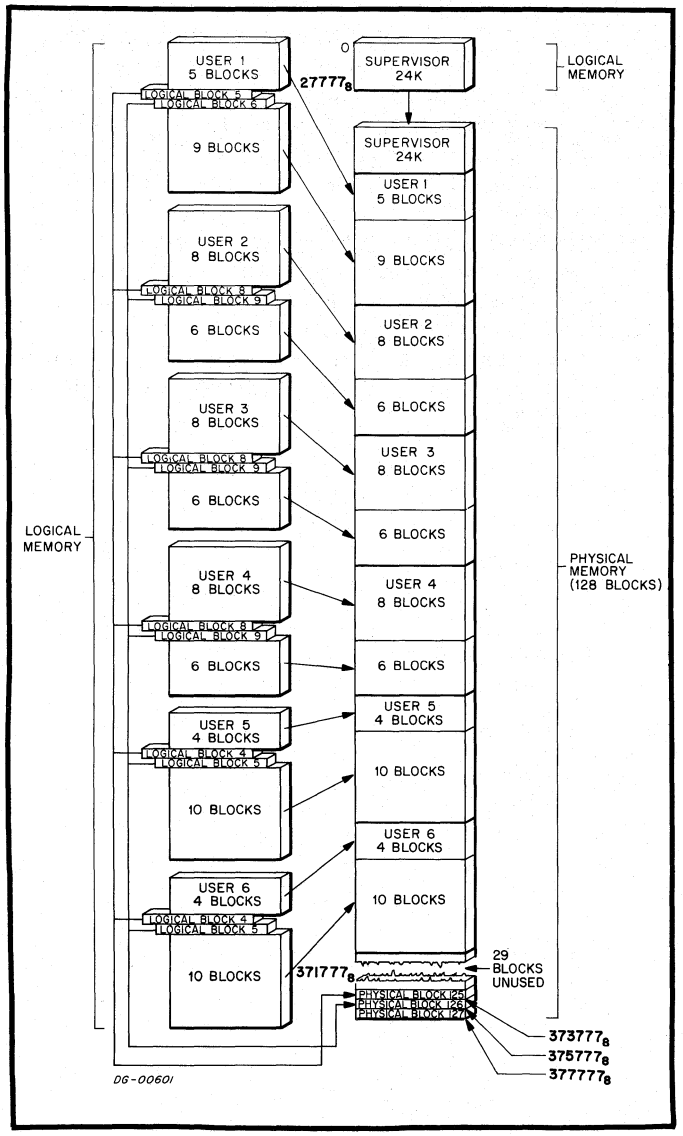
Given the same example of the nine users each requiring 24K bytes of physical memory and the tenth user requiring 30K, the ability to specify a different address computation for each block of logical memory means that as soon as any of the first nine users finishes his job, the tenth user can be serviced. Assume that user 2 finishes first. The supervisor could allocate these 24K bytes to hold the first 24K bytes of user 10's logical address space. The supervisor could then allocate 6K bytes of the 16K bytes left over from the first nine users to hold the last 6K bytes of user 10's logical address space.



In the preceding examples, it has been assumed that the only time address translation occurs is when the CPU requests a memory operation. In reality, both the CPU and the data channel can request memory operations. The MAP feature will accept logical addresses from both the CPU and the data channel, and then translate these addresses and perform the requested memory operation. If the MAP feature used the map for the current user to translate addresses for the data channel, then the only time a user could obtain the services of the data channel would be when that user was actually executing. In order to provide greater flexibility, the MAP feature allows the supervisor to specify a separate map for the data channel. This means that the data channel can service a user that is not the currently executing user. This allows the I/O activity of one user to be overlapped with the execution of another user.

By allowing a separate map to be specified for each block of logical memory, the MAP feature allows physical memory to be shared among users. Assume that six users are being serviced and that all the users are using a standard routine to perform some complicated computation. Further assume that this routine requires 4K bytes to run. If memory could not be shared, six copies of the same routine would have to be in physical memory at the same time. However, if the routine were written in such a manner that it did not modify itself, and if memory could be shared, only one copy would be needed. This would cut the physical memory requirements of this routine from 24K bytes to 4K bytes.

Sharing of physical memory is accomplished with the MAP feature simply by allocating the same block of physical memory to hold multiple blocks of logical memory. Assume that user 1 requires this computation to be in blocks 5 and 6 of his logical memory. Users 2, 3, and 4 require this computation to be in blocks 8 and 9 of their logical memory. Users 5 and 6 require this computation to be in blocks 4 and 5 of their logical memory. Now assume that the supervisor allocates physical blocks 125 and 126 to be used by this common routine. All the supervisor has to do to enable all the users to share this routine is map logical blocks 5 and 6 of user 1 to physical blocks 125 and 126, respectively; map logical blocks 8 and 9 of users 2, 3, and 4 to physical blocks 125 and 126; and map logical blocks 4 and 5 of users 5 and 6 to physical blocks 125 and 126. By doing this, 20K bytes are made available to service other users.



If the supervisor had to completely specify the map for a user each time that user was to receive the services of the CPU, the overhead time required would be substantial. The MAP feature can hold two user maps plus the map for the data channel at one time. This means that the supervisor can specify two user maps plus a data channel map at one time and then service these two users by disabling one map and enabling the other map. The data channel map can be enabled or disabled at any time.

MAP Protection Features

In addition to translating addresses, the MAP feature provides five different kinds of protection. These are validity protection, write protection, indirect protection, I/O protection, and data channel protection. Validity protection is always enabled. The other four protection features can be selectively enabled or disabled by the supervisor.

Validity protection protects the physical memory allocated to either the supervisor or a user from being accessed or altered by another user. If a user only requires 30K bytes to run his program, then the supervisor only allocates 30K bytes of physical memory to that user. This leaves 34K bytes of the user's logical address space unaccounted for. If the user's program is well written, it should never try to access one of these 34K bytes. Mistakes do happen, however, and if the user tries to access a location in logical memory, it is important that no harm will be done to either the supervisor or to other users.

In order to implement validity protection, the supervisor must specify to the MAP feature which logical addresses are to be declared invalid for each user. The supervisor does this by allocating enough physical memory to hold the amount of logical memory that the user says that he needs. Then, all remaining blocks of the user's logical address space are declared to be invalid. The supervisor declares a logical block to be invalid by mapping it to physical block 127 and declaring it to be write protected. If the MAP feature tries to translate an address for a user and finds that the logical address is invalid, a protection fault occurs.

Write protection allows the supervisor to ensure that certain blocks of allocated physical memory will not be altered. In the example of shared physical memory, it would be disastrous if one of the users altered a location in the shared routine. For this reason, the supervisor would probably declare

all the logical blocks mapped to physical blocks 125 and 126 to be write protected. The supervisor can write protect blocks of logical memory on a block-by-block basis. If write protection is enabled for a user, the MAP feature monitors all requests to modify memory. If the MAP feature detects a modify memory request and the logical address is in a block of logical memory that is write protected, a protection fault occurs.

Indirect protection allows the supervisor to ensure that the CPU will not be placed in an indirection loop. An indirection loop is the case where the effective address calculation follows a chain of indirect addresses and never fetches a word with bit 0 set to 0. When this happens, the effective address calculation never finishes and the CPU cannot finish the instruction.

To prevent the CPU from becoming disabled by a user indirection loop, the supervisor can enable indirect protection. With indirect protection enabled, the MAP feature monitors all indirect references. If the MAP feature detects 15 consecutive indirect references, it assumes that the chain of indirect address will never end and a protection fault occurs.

I/O protection allows the supervisor to protect the I/O devices in the system from unauthorized access. Devices can be declared accessible or inaccessible to a user on a device-by-device basis. With I/O protection enabled, the MAP feature monitors all I/O instructions. If the MAP feature detects an I/O instruction that refers to a device that has been declared inaccessible for this user, a protection fault occurs.

In lieu of I/O protection, the supervisor can enable the LOAD EFFECTIVE ADDRESS instruction for a user. If the LOAD EFFECTIVE ADDRESS instruction is enabled, then all instructions in the I/O format become LOAD EFFECTIVE ADDRESS instructions. The user cannot access any I/O device while LOAD EFFECTIVE ADDRESS is enabled.

Data channel protection allows the supervisor to write protect a block or blocks of logical memory in the data channel's logical address space. With data channel protection enabled, the MAP feature monitors all modify memory requests from the data channel. A modify memory request from the data channel is equivalent to a data channel input operation. If the MAP feature detects a modify memory request from the data channel and the logical address is in a block of logical memory that is data channel protected, the MAP feature does not perform the request and sets the data channel protection error bit in the MAP status register to 1. A protection fault does not occur and processing continues.

When the MAP feature detects a violation of any of the protection features that are enabled, it performs a protection fault. First the MAP feature disables the current user map. Then, it pushes a 5-word return block onto the stack that is defined by the stack control words found in physical locations $40\text{-}43_8$. The MAP feature then performs a "jump indirect" to location 3. This is a "jump indirect" to the address contained in physical location 3 which is the address of the supervisor's protection fault routine.

Due to the fact that the MAP feature can perform a protection fault at any point within the execution of an instruction, the return address placed in the fifth word of the return block is not always correct. For I/O protection violations, the return address is always the logical address of the instruction after the I/O instruction that caused the fault. For violations of validity protection, write protection, and indirect protection, the return address is either the logical address of the instruction that caused the fault, the logical address of the instruction after the instruction that caused the fault, or it is meaningless. If the MAP feature faults at a point within the instruction when the program counter is undefined, the PC UNDEFINED bit in the MAP status word is set to 1.

The MAP feature operates in two modes called user mode and supervisor mode. In user mode, all memory requests coming from the CPU are translated using the current user map. Checking is also performed for all protection features that are enabled. In the supervisor mode, memory requests in the range $0\text{-}75777_8$ are not translated. This means that the first 31 blocks of the supervisor's logical address space reside in the first 31 blocks of physical memory. In supervisor mode, all memory requests in the range $76000_8\text{-}77777_8$ are translated using the special map for supervisor logical block 31. This allows the supervisor to access portions of user space while in supervisor mode without resorting to lengthy use of the ENABLE SINGLE CYCLE instruction. The data channel map can be enabled or disabled in either mode.

If an I/O interrupt occurs while the MAP feature is in the user mode, the user map for the current user is disabled, the logical address of the next instruction to be executed for the current user is placed in physical location 0, and a "JMP @1" instruction is performed. This is a "jump indirect" to the address contained in physical location 1 which is the address of the supervisor's I/O interrupt handler.

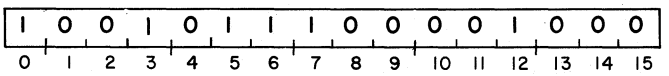
When power is first turned on, or after an I/O RESET instruction, the MAP feature is in the supervisor mode and the data channel map is disabled. Supervisor logical block 31 is mapped to physical block 31. On power up, the user maps, the data channel map, and the device protect codes are undefined.

MAP FEATURE INSTRUCTIONS

The MAP feature is programmed with a combination of I/O instructions and machine instructions. The instructions that affect the MAP feature are described on the following pages.

LOAD MAP

LMP



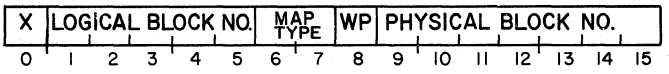
A BLOCK ADD AND MOVE instruction is performed with the exception that no data is written into the destination area. After the contents of AC0 have been added to the fetched word, the result is loaded into the MAP feature. The accumulators are set up in the same manner as for a BLOCK ADD AND MOVE instruction. If this instruction is issued while in the user mode, it is not executed if I/O protection is enabled. Program operation continues with the next sequential instruction.

Accumulator 3 is ignored and its contents remain unchanged.

The information to be loaded into the MAP feature is in three formats. Format number one defines the map for a single 2K byte block of logical memory. Format two defines the I/O devices that are inaccessible to a user. Format three defines the protection features that are to be enabled for a user.

Format Number One

Address Translation

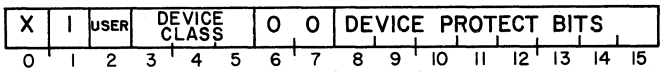


BIT NUMBER	CONTENTS
0	Unused
1-5	Logical block number--this is the number of the logical block that is to be mapped.
6-7	Map type-- if 01, this is a translation for the data channel-- if 10, this is a translation for user A if 11, this is a translation for user B.
8	Write protect-- if 1, this block may not be modified if write protection is enabled.
9-15	Physical block number--this is the number of the physical block that corresponds to the logical block given in bits 1-5.

NOTE A logical block is validity protected by mapping to physical block 127_{10} and setting the write protect bit.

Format Number Two

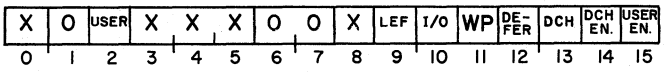
I/O Protection



BIT NUMBER	CONTENTS
0	Unused
1	Must be 1.
2	User number-- if 0, these devices are to be protected from user A; if 1, these devices are to be protected from user B.
3-5	Device class--this is an unsigned number in the range 0-7. This is the high-order digit of the two-digit octal device code.
6-7	Format type--must be 00.
8-15	Device protect bits--the second digit of the two-digit octal device code is specified by the position of the bit in this field. A 1 in any bit protects the corresponding device from receiving any commands directly from this user. For example, if bits 3-5 are 010 and bits 8-15 are 01010000, then devices 21_8 and 23_8 are protected.

Format Number Three

Status

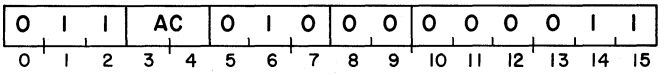


BIT NUMBER	CONTENTS
0	Unused
1	Must be 0.
2	User number-- if 0, these status bits refer to user A; if 1, these status bits refer to user B.
3-5	Unused
6-7	Format type--must be 00.
8	Unused
9	LEF mode-- if 1, the LOAD EFFECTIVE ADDRESS instruction is to be enabled for this user.
10	I/O protect-- if 1, I/O protection is to be enabled for this user.
11	Write protect-- if 1, write protection is to be enabled for this user.
12	Indirect protect-- if 1, indirect protection is to be enabled for this user.
13	Data channel protect-- if 1, data channel protection is to be enabled for this user.
14	Data channel map enable-- if 1, the data channel map is enabled immediately
15	User map enable-- if 1, the user map for this user is enabled after the LOAD MAP instruction is finished.

It is format three that directs the MAP feature to begin translating addresses. If at any time during the execution of the LOAD MAP instruction, the MAP feature receives a word in this format with bit 15 set to 1, the interrupt system is immediately disabled and the map for the user indicated by bit 2 is readied. After the next POP BLOCK, POP PC AND JUMP, RETURN, or RESTORE instruction or an indirect reference while computing an effective address, the map for the user indicated by bit 2 is enabled. After the first user instruction has started to execute, the interrupt system is enabled. The MAP feature will continue to translate addresses and check for protection violations until directed to stop by a SYSTEM CALL instruction or until it senses a protection violation, or an I/O interrupt occurs.

LOAD SINGLE WORD

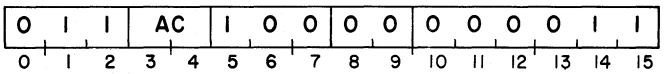
DOA ac, MAP



The contents of the specified AC are transferred to the MAP feature. The contents of the specified AC must be in one of the formats listed under the LOAD MAP instruction. The contents of the specified AC remain unchanged.

MAP SUPERVISOR BLOCK 31

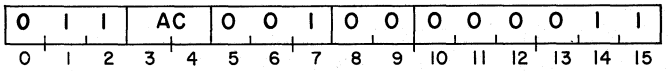
DOB ac, MAP



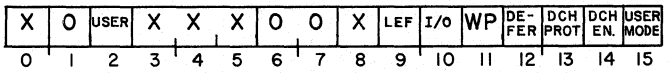
Bits 9-15 of the specified AC are transferred to the MAP feature. These bits specify a physical block number to which logical block 31 will be mapped when in the supervisor mode.

READ USER STATUS

DIA ac, MAP



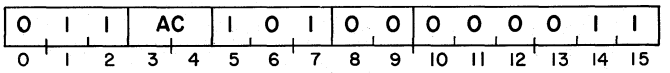
The status of the last enabled user map is placed in the specified AC. The previous contents of the specified AC are lost. The information placed in the specified AC has the following format:



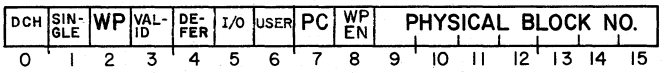
BIT NUMBER	CONTENTS
0	Unused
1	Always 0.
2	User number-- if 0, these status bits refer to user A if 1, these status bits refer to user B.
3-5	Unused
6-7	Always 0.
8	Unused
9	LEF mode-- if 1, the LOAD EFFECTIVE ADDRESS is enabled for this user.
10	I/O protect-- if 1, I/O protection is enabled for this user.
11	Write protect-- if 1, write protect is enabled for this user.
12	Indirect protection-- if 1, indirect protection is enabled for this user.
13	Data channel protect-- if 1, data channel protection is enabled for this user.
14	Data channel map enable-- if 1, the data channel is currently enabled.
15	User mode interrupt-- if 1, the last I/O interrupt occurred while in user mode.

READ MAP STATUS

DIC ac, MAP



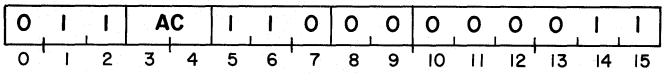
The contents of the MAP status register are placed in the specified AC. The previous contents of the specified AC are lost. The format of the information placed in the specified AC is as follows:



BIT NUMBER	CONTENTS
0	If 1, a data channel protection error has occurred.
1	If 1, the error occurred during a MAP SINGLE CYCLE instruction.
2	If 1, a write protection error has occurred for the user indicated in bit 6.
3	If 1, a validity protection error has occurred for the user indicated in bit 6.
4	If 1, an indirect protection error has occurred for the user indicated in bit 6.
5	If 1, an I/O protection has occurred for the user indicated in bit 6.
6	If 0, the last user map enabled was for user A if 1, the last user map enabled was for user B.
7	If 1, the program counter pushed onto the system stack is undefined.
8	If 1, write protection is enabled for the physical block whose number is given in bits 9-15.
9-15	This is the physical block number corresponding to the logical page number given in the last TRANSLATE BLOCK instruction.

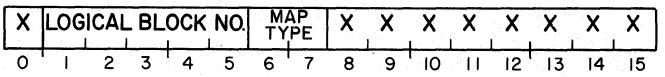
TRANSLATE BLOCK

DOC ac,MAP



The logical block number in bits 1-5 of the specified AC will be translated to the corresponding physical block number and placed in bits 9-15 of the MAP status register. The contents of the specified AC remain unchanged.

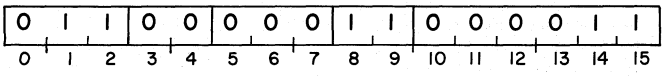
The format of the specified AC is as follows:



BIT NUMBER	CONTENTS
0	Unused
1-5	Logical block number to be translated.
6-7	If 00, no translation will be performed if 01, translation will be performed with the map for the data channel if 10, translation will be performed with the map for user A if 11, translation will be performed with the map for user B.
8-15	Unused

MAP SINGLE CYCLE

NIOP MAP



The last user map enabled is enabled for one memory reference. The first memory reference after the next LOAD or STORE instruction is mapped. After the memory cycle is mapped, the user map is again disabled.

Example:

If AC2 contains 405_8, and the following instruction sequence is issued:

```
NIOP MAP
LDA 3,2,2
```

The logical address 407_8 will be mapped using the user map for the last enabled user. The word contained in the corresponding physical location will be placed in AC3.

However, if the following instruction sequence is issued:

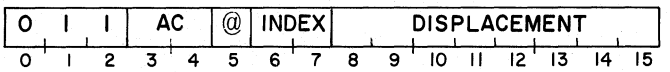
```
NIOP MAP
LDA 3,@2,2
```

The logical address 407_8 will be mapped using the user map for the last enabled user. The contents of the corresponding physical location will be used as the first level of an indirection chain. The next memory cycle, which is the second level of the indirection chain, will not be mapped.

NOTE The interrupt system is disabled from the beginning of the MAP SINGLE CYCLE instruction until after the next LOAD or STORE instruction.

LOAD EFFECTIVE ADDRESS

LEF ac,<@>displacement<,index>



If the LEF MODE bit in the user status is 1 for a user, then all I/O instructions issued by that user will be interpreted as LOAD EFFECTIVE ADDRESS instructions.

When a LOAD EFFECTIVE ADDRESS instruction is issued, the logical effective address is computed from bits 5-15 of the instruction and placed in the specified AC. The previous contents of the specified AC are lost. If an auto-incrementing or auto-decrementing location is referenced in the course of the effective address calculation, it is incremented or decremented.

Examples:

INSTRUCTION	RESULT
LEF 0,TABLE	The logical address of TABLE is placed in AC0.
LEF 2,34,2	34_8 is added to the unsigned integer in AC2.
LEF 1,-55,3	55_8 is subtracted from the unsigned integer in AC3 and the result is placed in AC1.
LEF 0,.+0	The logical address of this LOAD EFFECTIVE ADDRESS instruction is placed in AC0.

NOTE The LOAD EFFECTIVE ADDRESS instruction can only be issued while in the user mode.

DG OFFICES

SALES AND SERVICE OFFICES

Alabama: Birmingham
Arizona: Phoenix, Tucson
Arkansas: Little Rock
California: El Segundo, Fresno, Los Angeles, Oakland, Palo Alto, Pasadena, Sacramento, San Diego, San Francisco, Santa Ana, Santa Barbara, Van Nuys
Colorado: Denver, Englewood
Connecticut: North Branford, Norwalk
Florida: Ft. Lauderdale, Orlando, Tampa
Georgia: Norcross
Idaho: Boise
Iowa: Cedar Rapids Bettendorf
Illinois: Arlington Heights, Champaign, Chicago, Peoria, Rockford, Schaumburg
Indiana: Indianapolis
Kentucky: Louisville
Louisiana: Baton Rouge, Metairie
Maine: Portland
Maryland: Baltimore
Massachusetts: Cambridge, Springfield, Wellesley, Worcester
Michigan: Grand Rapids, Southfield
Minnesota: Richfield
Missouri: Creve Coeur, Kansas City, St. Louis
Mississippi: Jackson
Montana: Billings
Nebraska: Omaha
Nevada: Las Vegas, Reno
New Hampshire: Bedford, Nashua
New Jersey: Cherry Hill, Somerset, Wayne
New Mexico: Albuquerque
New York: Albany, Buffalo, Lake Success, Latham, Melville, Newfield, New York, Rochester, Syracuse, White Plains
North Carolina: Charlotte, Greensboro, Greenville, Raleigh
Ohio: Brooklyn Heights, Cincinnatti, Columbus, Dayton
Oklahoma: Oklahoma City, Tulsa
Oregon: Lake Oswego, Portland
Pennsylvania: Blue Bell, Carnegie, Lancaster, Philidelphia, Pittsburgh
Rhode Island: Providence, Rumford
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, Kirkland, Richland, Spokane
West Virginia: Charleston
Wisconsin: Brookfield, Madison, West Allis

INTERNATIONAL SUBSIDIARIES

Australia: Adelaide, Melbourne, New Castle, Sydney, Tasmania, Queensland, Victoria
Brazil: Sao Paulo
Canada: Calgary, Edmonton, Montreal, Ottawa, Quebec, Toronto, Vancouver, Winnipeg
France: Lille, Lyon, Nantes, Paris
Italy: Florence, Milan, Padua, Rome, Tourin
Japan: Tokyo
The Netherlands: Amsterdam, Rijswijk
New Zealand: Auckland, Wellington
Sweden: Gothenburg, Malmoe, Stockholm
Switzerland: Lausanne, Zurich
United Kingdom: Birmingham, Bristol, Chesire, Glasgow, Hounslow, London, Manchester
West Germany: Dusseldorf, Filderstadt, Frankfurt, Hamburg, Hannover, Munich, Nuremburg, Munich, Ratingen, Rodelheim, Stuttgart

DG-04976

REPRESENTATIVES & DISTRIBUTORS

Argentina: Buenos Aires
Bolivia: Novadata
Chile: Santiago
Columbia: Bogato
Costa Rica: San Jose
Ecuador: Quito
Egypt: Cairo
Finland: Helsinki
Guatemala: Guatemala City
Hong Kong: Hong Kong
India: Bombay
Indonesia: Jakarta
Israel: Givatayim
Korea: Seoul
Kuwait: Kuwait
Lebanon: Beirut
Malaysia: Kuala Lumpur
Mexico: Mexico City, Nuevo Leon
Morocco: Casablanca
Nicaragua: Managua
Nigeria: Ibadan, Lagos
Norway: Oslo
Paraguay: Asuncion
Peru: Lima
Philippine Islands: Manila
Portugal: Lisbon
Puerto Rico: Hato Rey
Saudi Arabia: Jaddah, Riyadh
Singapore: Singapore
South Africa: Capetown, Durban, Johannesburg, Pretoria
Spain: Barcelona, Bibao, Madrid, San Sebastian, Valencia
Taiwan: Taipei
Thailand: Bangkok
Turkey: Ankara
Uruguay: Montevideo
Venezuela: Maracaibo

ADMINISTRATION, MANUFACTURING RESEARCH AND DEVELOPMENT

Massachusetts: Cambridge, Framingham, Southboro, Westboro
Maine: Westbrook
New Hampshire: Portsmouth
California: Anaheim, Sunnyvale
North Carolina: Research Triangle Park, Johnston County
Texas: Austin

Hong Kong: Kowloon, Tai Po
Thailand: Bangkok

CUT ALONG DOTTED LINE

DataGeneral

users group

Installation Membership Form

Name _____ Position _____ Date _____

Company, Organization or School _____

Address _____ City _____ State _____ Zip _____

Telephone: Area Code _____ No. _____ Ext. _____

1. Account Category

- ☐ OEM
- ☐ End User
- ☐ System House
- ☐ Government
- ☐ Educational

2. Hardware

	Qty. Installed	Qty. On Order
M/600		
COMMERCIAL ECLIPSE		
SCIENTIFIC ECLIPSE		
AP/130		
CS Series		
Mapped NOVA		
Unmapped NOVA		
microNOVA		
Other (Specify) _____		
_____		
_____		

3. Software

- ☐ AOS
- ☐ DOS
- ☐ MP/OS
- ☐ RDOS
- ☐ Other

Specify _____

4. Languages

- ☐ Algol
- ☐ DG/L
- ☐ Cobol
- ☐ PASCAL
- ☐ Business BASIC
- ☐ BASIC
- ☐ Assembler
- ☐ Fortran
- ☐ RPG II
- ☐ PL/1
- ☐ Other

Specify _____

5. Mode of Operation

- ☐ Batch (Central)
- ☐ Batch (Via RJE)
- ☐ On-Line Interactive

6. Communications

- ☐ HASP
- ☐ RJE80
- ☐ RCX 70
- ☐ CAM
- ☐ XODIAC
- ☐ Other

Specify _____

7. Application Description

○ _____

8. Purchase

From whom was your machine(s) purchased?

- ☐ **Data General Corp.**
- ☐ Other

Specify _____

9. Users Group

Are you interested in joining a special interest or regional Data General Users Group?

○ _____

DG-06893

DataGeneral

Data General Corporation, Westboro, Massachusetts 01580, (617) 366-8911

FOLD FOLD

FOLD FOLD

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:

Data General

ATTN: Users Group Coordinator
4400 Computer Drive
Westboro, MA **01580**

CUT ALONG DOTTED LINE

DataGeneral

Engineering Publications Comment Form

Please help us improve our future publications by answering the questions below. Use the space provided for your comments.

Title: _____

Document No. 014-000626-02

Yes	No		
☐	☐	Is this manual easy to read?	○ You (can, cannot) find things easily. ○ Language (is, is not) appropriate. ○ Technical terms (are, are not) defined as needed. ○ Other:
		In what ways do you find this manual useful?	○ Learning to use the equipment ○ As a reference ○ As an introduction to the product ○ To instruct a class. ○ Other:
☐	☐	Do the illustrations help you?	○ Visuals (are,are not) well designed. ○ Labels and captions (are,are not) clear. ○ Other:
☐	☐	Does the manual tell you all you need to know? What additional information would you like?	
☐	☐	Is the information accurate? (If not please specify with page number and paragraph.)	

Name: _____ Title: _____

Company: _____ Division: _____

Address: _____ City: _____

State: _____ Zip: _____ Telephone: _____ Date: _____

DG-06895

DataGeneral
Data General Corporation, Westboro, Massachusetts 01580

FOLD FOLD

FOLD FOLD

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:

Data General
ATTN: ENGINEERING PUBLICATIONS
4400 Computer Drive
Westboro, MA 01580

Data General

Data General Corporation, Westboro, Massachusetts 01580