ECLIPSE® C/150

Principles of Operation

Data General

ECLIPSE® C/150

PRINCIPLES OF OPERATION

Data General

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.

NOVA, INFOS, and **ECLIPSE** are registered trademarks of Data General Corporation, Westboro, Massachusetts. **DASHER**, **microNOVA**, and **ECLIPSE MV/8000** are trademarks of Data General Corporation, Westboro, Massachusetts.

Ordering No. 014-000616

Printed in the United States of America
Rev. 01, September 1980

CONTENTS

CHAPTER I ECLIPSE C/150 SYSTEM

CHAPTER II CONCEPTS AND FACILITIES

CONTENTS CONTINUED

CONTENTS CONTINUED

Chapter I
ECLIPSE C/150 SYSTEM

Data General's ECLIPSE C/150 is ideal for applications where commercial instruction capability is required but where a large computer is unnecessary.

HIGHLIGHTS OF THE ECLIPSE C/150 SYSTEM

Despite its small size, the ECLIPSE C/150 can execute the entire ECLIPSE standard instruction plus the ECLIPSE Floating Point Instruction Set, the Character Instruction Set (CIS), and the ECLIPSE Decimal/Edit instruction set. As such, it is ideal for applications where a large computer is unnecessary but full commercial instruction capability is required.

In this section, we discuss some of the features of the C/150 which are important to the system programmer.

Main Storage

The ECLIPSE C/150 has a maximum memory capacity of 256 Kbytes. Minimum memory is 128 Kbytes. Both Error Checking and Correcting (ERCC) semiconductor memory and core memory are available.

Semiconductor memory modules can be 2-way interleaved; core memory cannot be interleaved.

The C/150 Memory Allocation and Protection unit protects individual user space within memory on a 2 Kbyte page basis. Protection modes include address validity, infinite defer, write, and I/O protection.

I/O Management System

The ECLIPSE C/150 has several systems for transferring information to and from the computer. Each method is appropriate for certain types of peripherals.

The standard NOVA/ECLIPSE data channel provides I/O communication for medium- and high-speed devices such as disc drives, magnetic tapes drives, data channel line printers, and synchronous communications devices. Maximum transfer rates are 2.5 Mbyte/second input, 1.7 Mbyte/second output.

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

Main Processor

The ECLIPSE C/150 main processor can execute the standard ECLIPSE instruction set, using a fast, hardware-assisted, integer multiply/divide function. The standard 56-instruction ECLIPSE floating point instruction set is also implemented in the C/150 firmware.

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

The Decimal/Edit instruction set handles many types of commercial operations, using a variety of industry-compatible formats. In addition, the Edit subprogram can perform many different operations on a decimal number, including leading zero suppression, floating fill characters, punctuation control, and insertion of text into the destination field.

Software Support

A wide variety of software support is available for the ECLIPSE C/150 system.

The Real-Time Disc Operating System (RDOS) supports multi-terminal interactive operations in a foreground/background multi-tasking environment. It also supports INFOS^{TM} (Data General's sophisticated file management system) and, with INFOS^{TM}, the Idea (Interactive Data Entry and Access) system for multi-terminal transaction processing.

Many higher-level languages are available with RDOS, including COBOL, RPG II, and Business BASIC. In addition, RDOS supports Fortran IV, Fortran V, Extended Basic, DG/L^{TM} (an ALGOL-derivative structured programming language), and Macro Assembler. In addition, RDOS supports COBOL, RPG II, and Business BASIC.

The Advanced Operating System (AOS) uses adaptive resource management for efficient operation in multiuser environments. It supports concurrent batch, timesharing, and real-time operations. INFOS ^{TM}, COBOL, RPG II, and Idea are not available with AOS.

Chapter II
CONCEPTS AND FACILITIES

The ECLIPSE C/150 contains a variety of extremely powerful standard ECLIPSE facilities, including:

ECLIPSE standard instruction set,
stack,
data channel,
MAP,
character instruction set (CIS),
decimal arithmetic instructions,

In this chapter we describe the facilities which are standard on all ECLIPSE C/150s, and the assembly-language instructions which control these facilities.

You can find complete descriptions of all the ECLIPSE C/150 assembly-language instructions, other than I/O instructions, in Chapter III. Chapter IV contains complete descriptions of all the I/O instructions.

ADDRESSING CONVENTIONS

The various methods of addressing memory locations in the ECLIPSE C/150 give you considerable flexibility when storing and retrieving data, or transferring control to a different procedure.

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

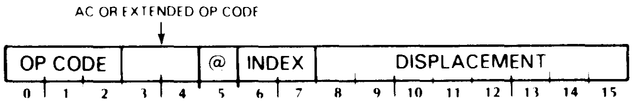
In this manual, we speak of a user's *address space* of 15 bits. This is a reference to the *logical* address space; the address space the user normally sees, which can be addressed by a 15-bit address. The maximum amount of logical address space available to the programmer is 32,768 words. (The *physical* address space - corresponding to the total amount of main memory in the computer - may be much larger.) Within a logical address space, the next sequential memory location after location 77777_8 is location 0.

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

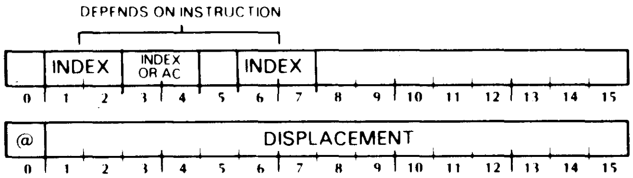
Word Addressing Definitions

The following definitions are useful for understanding word addressing in the ECLIPSE C/150:

SHORT CLASS



EXTENDED CLASS



Addressing Modes - Methods of addressing using a displacement from some reference point to find the desired address. There are three different modes, each using different reference points.

Indirect Addressing - A method of addressing which uses the first address found as a pointer to another address which, in turn, may be used as a pointer to yet another address, etc. A series of indirect addresses is called an *indirection chain.*

Index Bits - Bits in the instruction which control the addressing mode used when executing this instruction.

Indirect Bit - A bit in the instruction or address which controls the indirection chain at each step of the addressing process.

Displacement Bits - Bits in the instruction which control the displacement distance, in memory locations, between some reference point (determined by the mode) and the desired address.

Effective Address Calculation - Logical process of converting the index, indirect, and displacement bits into an address to be used by the instruction.

Intermediate Address - The address obtained by the effective address calculation before testing for indirection.

Lower Page Zero - Locations 0-377_8 in memory.

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

INDEX BITS	RANGE OF DISPLACEMENT FIELD	
	SHORT CLASS	EXTENDED CLASS
00	0 to 377_8 or 0 to 255_{10}	0 to 77777_8 or 0 to $32,767_{10}$
01 10 11	-200_8 to 177_8 or -128 to +127_{10}	-40000_8 to 37777_8 or -16,384 to +$16,383_{10}$

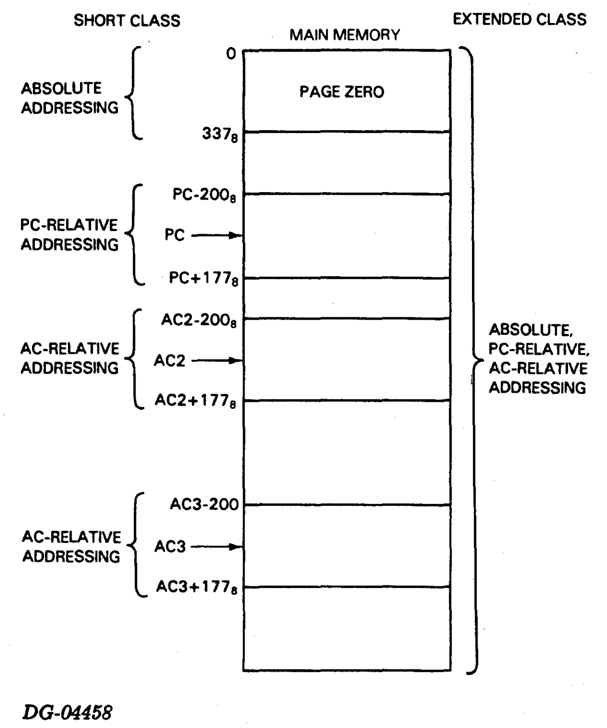
Addressing Modes

Word addressing in the ECLIPSE C/150 can be done in the following modes:

- absolute addressing;
- P.C. (program counter) relative addressing;
- accumulator relative addressing.

In addition, direct or indirect addressing can be used in any of these modes. By choosing the proper mode at the appropriate time, you can obtain access to any address in your logical address space.

The figure below illustrates the three addressing modes.



Absolute Addressing Mode - In absolute addressing mode, the intermediate address is set equal to the unmodified displacement. As a result, the short class of instructions specify locations in the range 0-377_8 in the absolute mode (short class instructions are restricted to 8 bits in the displacement).

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

Extended class instructions can reference any logical memory address using the absolute addressing mode.

P.C. Relative Addressing Mode - In P.C. relative addressing mode, the intermediate address is found by adding the displacement to the address of the word containing the displacement.

Accumulator Relative Addressing Mode - In accumulator relative addressing mode, the intermediate address is found by adding the displacement to the contents of the accumulator indicated by the index bits (you may use either AC2 or AC3).

Direct and Indirect Addressing - Direct addressing uses the intermediate address without modification.

Indirect addressing uses the intermediate address as a pointer to the next address. If bit 0 of the next address is 1, this address is used as a *pointer* which points to another address. The indirection chain is continued until an address is found with bit 0 equal to 0. This address is then used as the address of the data.

Any number of indirection levels is permitted in the ECLIPSE C/150, but indirect protection is available which can limit indirections to 15 levels (see the MAP section).

Auto-Incrementing and Auto-Decrementing - If the intermediate address of a short class instruction is in the range 20-27$_8$, and the indirect bit is 1, the contents of the addressed location are incremented by one, and the addressing chain continues using the *incremented* value of the addressed location.

If the intermediate address of a short class instruction is in the range 30-37$_8$, and the indirect bit is 1, the contents of the addressed location are decremented by one, and the addressing chain continues using the *decremented* value of the addressed location.

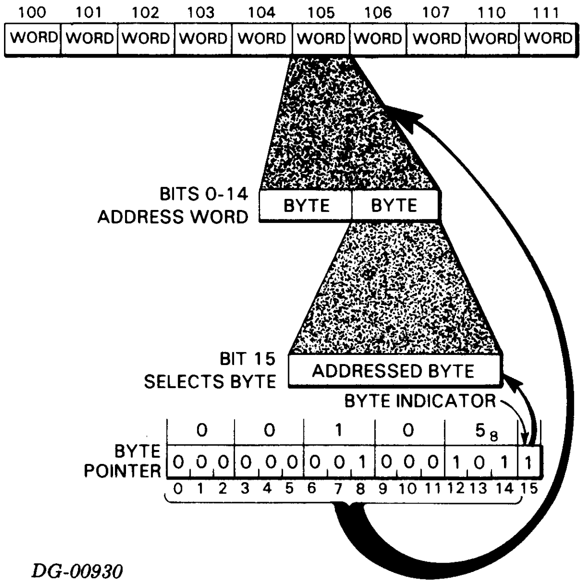
> NOTE: *The state of bit 0* before *the increment or decrement determines whether the indirection chain is continued. For example: Assume an auto-increment location contains 177777$_8$ (all bits = 1 including bit 0), and the location is referenced as part of an indirection chain. After incrementing, the location contains all zeros. However, bit 0 was 1 before the increment, so 0 will be the next address in the chain, rather than the effective address.*

You can find a flow diagram of the addressing process in a Appendix F.

BYTE MANIPULATION

Byte Format

We represent bytes as 8-bit unsigned binary integers. A byte in memory is selected by a 16-bit *byte pointer*. Bits 0-14 of the byte pointer contain the memory address of a 2-byte word. Bit 15 (the *byte indicator*) indicates which byte of the addressed location will be used. If bit 15 is 0, the high-order byte (bits 0-7) will be used. If bit 15 is 1, the low-order byte (bits 8-15) will be used. See the figure below.



DG-00930

BYTE INSTRUCTIONS

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

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

Byte Instructions

Mnem	Name	Function
LDB ELDB	Load Byte	Places a byte of information into an accumulator.
STB ESTB	Store Byte	Stores the right byte of an accumulator into a byte of memory.

BIT MANIPULATION

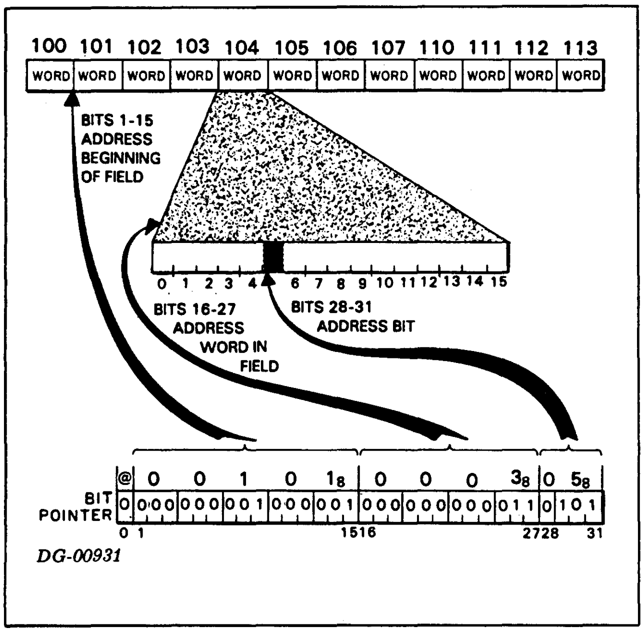
Bit Addressing

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

We form the address of the desired bit as follows:

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

Below is a diagram of the bit-addressing process.



DG-00931

BIT INSTRUCTIONS

The ECLIPSE C/150 instructions which manipulate bits:

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

Some of the bit instructions use a bit pointer to locate a bit in memory. The others only affect bits within the specified accumulators.

BIT MANIPULATION INSTRUCTIONS

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

CHARACTER MANIPULATION

Character Instructions

The four character instructions manipulate strings of characters. Each character in a string occupies one byte. These strings can be any data type. The character instructions:

- compare one byte string to another;
- move a byte string from one area of memory to another;
- translate a character string from one data type to another.

The character instructions are described in the table below.

CHARACTER INSTRUCTIONS

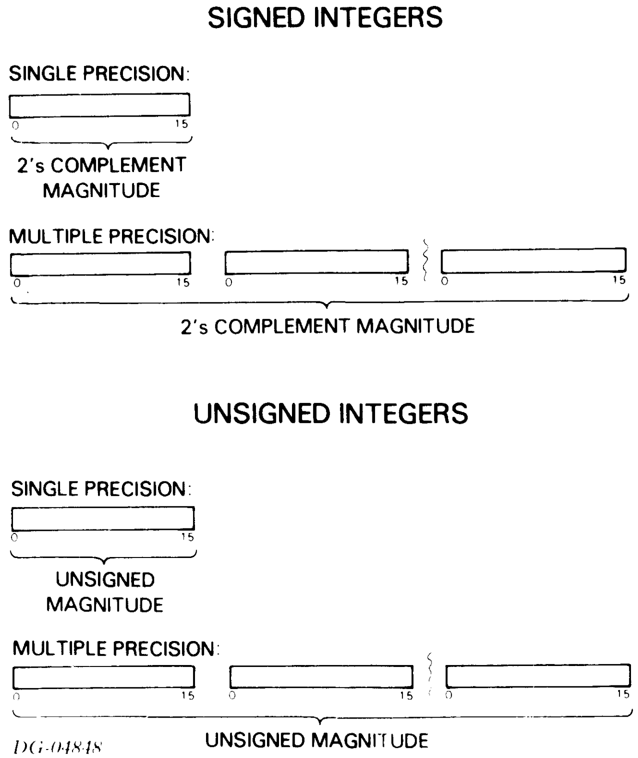
Mnem	Name	Function
CMP	Character Compare	Compares one string of characters in memory to another string.
CMT	Character Move Until True	Moves a string of bytes from one area of memory to another until a table-specified delimiter character is encountered or the source string is exhausted.
CMV	Character Move	Moves a string of bytes from one area of memory to another under control of the values in the four accumulators.
CTR	Character Translate	Translates a string of bytes from one data representation to another and either moves it to another area of memory or compares it to a second string of bytes.

NUMBER CONVENTIONS

Integer Format

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

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



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

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

In addition, there is a *Carry* bit. A change in the value of the carry bit indicates a carry out during fixed point arithmetic operations.

Decimal Format

We represent decimal numbers by a variety of industry-compatible formats. Both *unpacked* and *packed* decimal format can be recognized and manipulated by various instructions.

Unpacked Decimals

In unpacked decimal format, each byte of memory contains the code for one ASCII character. Each decimal digit is represented by the ASCI character for that digit except when a digit and sign are combined in one character. The table below shows the ASCII characters we use to represent the combination of a digit and sign in those formats which require it.

Digit	Digit With + Sign		Digit With - Sign	
	ASCII Character	Octal	ASCII Code	Octal Character Code
0	{	173	}	175
1	A	101	J	112
2	B	102	K	113
3	C	103	L	114
4	D	104	M	115
5	E	105	N	116
6	F	106	O	117
7	G	107	P	120
8	H	110	Q	121
9	I	111	R	122

You can represent the sign in any one of four ways when using unpacked decimal format. These four ways are shown in the table that follows.

Note that in each example, the first line shows the decimal number as normally written, the second line shows the ASCII characters placed in each byte, and the third line shows the octal code of the character in each byte.

Type	Characteristic	Example
Leading Sign	Sign appears in separate byte after number.	+2048 + 2 0 4 8 053 062 060 064 070
Trailing Sign	Sign appears in seperate byte after number.	-1756 1 7 5 6 - 061 067 065 066 055
High-order Sign	Sign and high-order digit are indicated by single (first) byte.	+1850 A 8 5 0 101 070 065 060
Low-order Sign	Sign and low-order digit are indicated by single (last) byte.	-3972 3 9 7 K 063 071 067 113

Packed Decimal

In packed decimal format, each digit of the decimal number occupies one half byte in memory. The sign appears in a separate trailing half byte. The number must start and end on a byte boundary, so a packed decimal number always consists of an odd number of digits followed by the sign (a zero is placed in front of numbers with an even number of digits). The sign is represented by the octal number 14_8 for plus and 15_8 for minus.

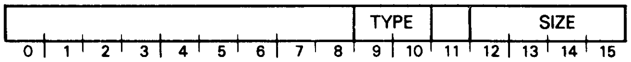
Several examples of packed decimal numbers are shown below.

	BYTE		BYTE		BYTE	
+2048	0 00	2 02	0 00	4 04	8 10	+ 14
+32,456	3 03	2 02	4 04	5 05	6 06	+ 14
-1756	0 00	1 01	7 07	5 05	6 06	- 15
-25,989	2 02	5 05	9 11	8 10	9 11	- 15

Data Type Indicator

Most ECLIPSE C/150 instructions make certain assumptions about the representation of data in memory -- whether the data you are referencing is in integer format, floating point format and so on. The assumptions about data type made by the instructions are usually obvious; your choice of instruction implicitly defines the kind of data you are manipulating. For example the *Load byte* assumes the information to which you refer is a single byte of data, while the *Load floating point double* instruction operates on an aggregate of data in memory that is eight bytes long.

However, the decimal arithmetic and the edit instructions do not make such assumptions. Rather, these instructions require that you pass them a parameter called the *data-type indicator*, which defines both the data representation you want the operation to use, and also its size. You pass the indicator in an accumulator. The data-type indicator has the following format:



BITS	NAME	CONTENTS or FUNCTION
0-7	---	Reserved for future use
8-10	TYPE	Data type: 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
11-15	SIZE	Data length: For all except data type 5, count of bytes in number *minus 1* (including sign); For data type 5, the count of *digits* in the number

Logical Format

We represent logical entities as individual bits in a 16-bit word. Each bit is treated as a separate binary value. When two words are involved (logical AND or XOR, for example) only corresponding bits of each word interact. Examples of logical operations include:

- forming the logical AND of two words;
- forming the logical complement of a word;
- shifting the contents of a word left or right.

Floating Point Format

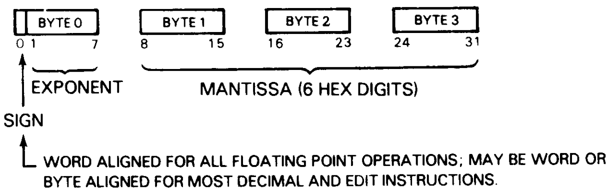
Word for word, floating point format provides a much larger range than integer format, at the expense of some precision. It also provides the ability to operate on fractions. The maximum range of floating point format is equivalent to a 16-word multiple precision integer. In addition, floating point operations are executed faster than most multiple precision integer operations.

We represent a floating point value using a 4-byte-wide (for single precision) or an 8-byte-wide (for double precision) number. The 4- or 8-byte aggregate contains 3 fields:

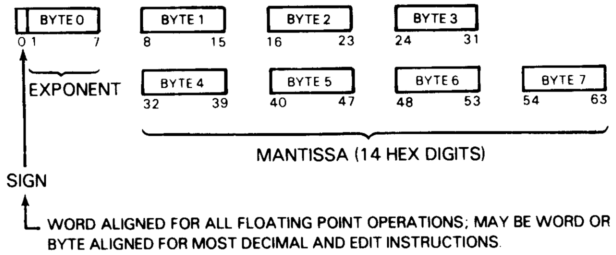
- a fractional part called the mantissa, which, at the end of all floating point mathematics operations, is always adjusted to be greater than or equal to 1/16 and less than 1 (i.e., *normalized*);
- an exponent, which is adjusted to maintain the correct value of the number;
- a sign.

Operations on numbers in memory employing the floating point arithmetic instructions require that the number be *word aligned*, so that bit 0 of the first byte of the number is bit 0 of first word of a 2-word or 4-word area in memory. Certain operations on numbers in memory employing decimal or edit instructions allow the number to be either word aligned or *byte aligned*. Byte alignment means that bit 0 of the first byte of the number is either bit 0 or bit 8 of any word in memory.

SINGLE PRECISION (4 BYTES)


BYTE 0
BYTE 1
BYTE 2
BYTE 3
0 1 7
8 15
16 23
24 31
EXPONENT
MANTISSA (6 HEX DIGITS)
SIGN
WORD ALIGNED FOR ALL FLOATING POINT OPERATIONS; MAY BE WORD OR BYTE ALIGNED FOR MOST DECIMAL AND EDIT INSTRUCTIONS.

DOUBLE PRECISION (8 BYTES)



DG-04849

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

MANTISSA X $16^{\text{(TRUE VALUE OF THE EXPONENT)}}$

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

Sign

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

Exponent

The right-most 7 bits of the first byte contain the exponent. We use *excess 64* representation. For both positive and negative exponents, the value is 64 greater than the true value of the exponent. The following table illustrates this:

EXPONENT FIELD	TRUE VALUE of EXPONENT
0	-64
64	0
127	63

Mantissa

Bytes 1-3 (single precision) or bytes 1-7 (double precision) contain the mantissa. By definition, the binary point lies *between* byte 0 and byte 1 of a floating point number. In order to keep the mantissa in the range of 1/16 to 1, the results of each floating point calculation are *normalized.* A mantissa is normalized by shifting it left one hex digit (4 bits) at a time, until the high-order four bits (the left-most four bits of byte 1) represent a nonzero quantity. For every hex digit shifted, the exponent is decreased by one.

NUMBER MANIPULATION

Fixed Point Arithmetic Instructions

There are 26 ECLIPSE C/150 instructions which perform fixed point arithmetic. These instructions:

- Perform binary arithmetic on operands in accumulators;
- Load data from memory to an accumulator;
- Move data from an accumulator to memory;
- Load a number into an accumulator.

All of the fixed point arithmetic instructions are shown in the following table. Some of the instructions appear in both a short form and a long form (the long form is usually indicated by the prefix *E* in the mnemonic). Most of these are instructions that move data to or from memory. For these, the short form is 16 bits in length and can directly specify a memory address from 0 to 255, or can directly specify a small area in memory surrounding the present value of the program counter or an accumulator. Long form instructions are 32 bits in length; they can directly specify any address from 0 to 77777_8.

ADI and ADDI are also short and long forms, respectively, of the same instruction. The short form can only add a 2-bit quantity, coded with the instruction (an *immediate*) in the range 1-4, while the long form can add a 16-bit immediate in the range -32,768 to +32,767.

FIXED POINT INSTRUCTIONS

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

FIXED POINT INSTRUCTIONS Continued

Mnem	Name	Function
SBI	Subtract Immediate	Subtracts an unsigned integer in the range 1-4 from the contents of an accumulator.
STA, ESTA	Store Accumulator	Stores data in memory from an accumulator.
SUB	Subtract	Subtracts contents of one accumulator from another.
XCH	Exchange Accumulators	Exchanges the contents of two accumulators.

DECIMAL ARITHMETIC

There are 11 instructions in the ECLIPSE C/150 which perform operations on decimal data. These instructions:

- Add and subtract decimal integers;
- Shift the contents of words one or more hex digits left or right;
- Convert decimal integers to floating point numbers;
- Convert floating point numbers to decimal integers of a specified data type;
- Convert decimal integers to strings of bytes and perform a variety of functions on the string.

Decimal Faults

In the course of processing decimal instructions, the CPU performs certain checks on the data being processed. If an invalid data type or number is found, a fault is initiated. When a fault occurs, the processor first pushes a return block onto the stack, with the program counter word in the return block pointing to the instruction that caused the fault. It then places a code indicating the type of fault in AC1, and executes a *Jump indirect* to the decimal fault address, location 46_8. This location should point to a fault handling routine.

The table below describes the decimal faults:

CODE	INSTR.	MEANING
4	LDI STI STIX	Number too large to convert to specified data type. SI/DI is in AC2.
6	LSN LDI LDIX	Sign code is invalid for this data type. AC3 contains SI.
7	LSN LDI LDIX	Invalid digit. AC2 contains SI.

DECIMAL ARITHMETIC INSTRUCTIONS

Mnem	Name	Function
DAD	Decimal Add	Adds together the decimal digits found in bits 12-15 of two acculumulators.
DHXL	Double Hex Shift Left	Shifts the 32-bit contents of two accumulators left 1 to 4 hex digits.
DHXR	Double Hex Shift Right	Shifts the 32-bit contents of two accumulators right 1 to 4 hex digits.
DSB	Decimal Subtract	Subtracts the decimal digit in bits 12-15 of one accumulator from the decimal digit in bits 12-15 of another acumulator.
EDIT	Edit	Converts a decimal integer to a string of bytes controlled by an edit subprogram; or manipulates string of bytes.
HXL	Hex Shift Left	Shifts the contents of an accumulator left a number of hex digits.
HXR	Hex Shift Right	Shifts the contents of an accumulator right a number of hex digits.
LDI	Load Integer	Converts a decimal integer to normalized floating point form and places it in a specified floating point accumulator.
LDIX	Extended Load Integer	Distributes a decimal integer into 4 floating point accumulators.
LSN	Load	Evaluates a number in memory and returns a code indicating the sign of the number.
STI	Store Integer	Converts the contents of a floating point accumulator to a specified format and stores it in memory.
STIX	Extended Store Integer	Converts the contents of 4 floating point accumulators to integer form and uses the 8 low-order digits of each to form a 32-bit integer.

Logical Operation Instructions

All of the logical operations instructions are shown in the following table. The *Load Effective Address* and *Extended Load Effective Address* instructions are the short and long form, respectively, of the same instruction. The short form is 16 bits in length and can directly specify a memory address from 0 to 255 or can directly specify a small area in memory surrounding the present value of the program counter or an accumulator. Long form instructions are 32 bits in length; they can directly specify any address from 0 to 77777_8.

LOGICAL OPERATION INSTRUCTIONS

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

Floating Point Arithmetic

The ECLIPSE C/150 floating point instructions assume normalized input numbers. Results are undefined for unnormalized input.

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 at right.

Guard Digit

In order to increase accuracy, a 4-bit (1 hex digit) *guard digit* is used during floating point arithmetic operations. This guard digit accepts and holds up to 4 bits shifted out (to the right) of the mantissa, and is used in all single precision and double precision operations until the completion of each instruction. The guard digit is truncated before the data is stored at the end of the instruction process.

Floating Point Fault Conditions

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

- overflow
- underflow
- divide by zero
- mantissa overflow

ANY	OVF	UNF	DVZ	MOF	TE	Z	N					FPMOD			
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

FLOATING POINT PROGRAM COUNTER															
16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31

BITS	NAME	CONTENTS or FUNCTION
0	ANY	Indicates that any of bits 1-4 are set.
1	OVF	Overflow Indicator--while processing a floating point number, an exponent overflow occurred; the result is correct except the exponent is 128 too small.
2	UNF	Underflow Indicator - while processing a floating point number, an exponent underflow occurred; the result is correct except that the exponent is 128 too large.
3	DVZ	Divide by Zero - while processing a floating point number, a zero divisor was detected; division was aborted and the operands remain unchanged.
4	MOF	Mantissa Overflow - during a FSCAL instruction, a significant bit was shifted out of the high order end of the mantissa; this bit is also set during a FIX instruction if the result does not fit into the destination location.
5	TE	Trap Enable - If this bit is 1, setting any of bits 1-4 will result in a floating point fault.
6	Z	Zero bit - The result of the last floating point operation was zero.
7	N	Negative bit--The result of the last floating point operation was less than zero.
8-11*	---	Reserved for future use.
12-15	FPMOD	Indicates computer series supporting the floating point instruction set. 0000 S/200, C/300, S/230, C/330 0001 S/130, S/250 standard FP,C/150 0010 M/600, C/350, S/250 optional FP 0011 Reserved for future use.
16	---	Reserved for future use.
17-31	FPPC	Floating Point Program Counter - This is the logical address of the last floating point instruction executed. In the event of a floating point fault, this is the address of the floating point instruction that caused the fault.

**These bits are used as internal flags by the firmware; Preserve them when saving the state of the FPSR.*

Floating Point Trap

If the program has set bit 5 of the floating point status register to 1, a floating point fault condition will initiate a floating point trap. Immediately before the next floating point instruction is executed, a return block is pushed onto the stack and the program counter jumps indirect via location 45_8. Location 45_8 should contain the address of the floating point fault handler. The return block pushed has the following format:

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

NOTE: *The return address is not the address of the floating point instruction that caused the fault, nor is it (necessarily) the address of the instruction following the instruction that caused the fault. It is the address of the* floating point *instruction following the instruction that caused the fault.*

If the instruction following the instruction that caused the fault is a Push Floating Point State *or a* Pop Floating Point State, *the fault will not occur immediately. It will occur when the system returns to the same user environment and is about to execute a floating point instruction other than a* Push Floating Point State *or a* Pop Floating Point State . *In this way, the fault will only occur within the user environment which caused it.*

The floating point instructions are shown in the following table. Note that several instructions have two forms; one ending in *S* and one ending in *D*. The first form uses single-precision floating point format, while the second form uses double-precision floating point format. The function of the two forms is otherwise identical.

FLOATING POINT INSTRUCTIONS

Mnem	Name	Function
FAB	Absolute Value	Sets the sign bit of an FPAC to 0.
FAMS, FAMD	Add (memory to FPAC)	Adds the floating point number in memory to the floating point number in an FPAC.
FAS, FAD	Add (FPAC to FPAC)	Adds the floating point number in one FPAC to the floating point number in another FPAC.
FCLE	Clear Errors	Sets bits 0-4 of the FPSR TO 0.
FCMP	Compare Floating Point	Compares two floating point numbers and sets the Z and N flags accordingly.
FDMS, FDMD	Divide (FPAC by memory)	Divides the floating point number in an FPAC by a floating point number in memory.
FDS, FDD	Divide (FPAC by FPAC)	Divides the floating point number in one FPAC by the floating point number in another FPAC.
FEXP	Load Exponent	Places bits 1-7 of AC0 in bits 1-7 of the specified FPAC.
FFAS	Fix To AC	Converts the integer portion of a floating point number to a signed two's complement integer and places the result in an accumulator.
FFMD	Fix To Memory	Converts the integer portion of a floating point number to double-precision integer format and stores the result in two memory locations.
FHLV	Halve	Divides the floating point number in FPAC by 2.
FINT	Integerize	Sets the fractional portion of the floating point number in the specified FPAC to zero and normalizes the result.
FLAS	Float From AC	Converts a signed two's complement number in an accumulator to a single precision floating point number.
FLDS, FLDD	Load Floating Point	Moves a floating point number from memory to a specified FPAC.
FLMD	Float From Memory	Converts the contents of two memory locations in integer format to floating point format and places the result in a specified FPAC.
FLST	Load Floating Point Status	Moves the contents of two specified memory locations to the FPSR.
FMMS, FMMD	Multiply (memory by FPAC)	Multiplies the floating point number in memory by the floating point number in an FPAC.

FLOATING POINT INSTRUCTIONS (Continued)

Mnem	Name	Function
FMOV	Move Floating Point	Moves the contents of one FPAC to another FPAC.
FMS, FMD	Multiply (FPAC by FPAC)	Multiplies the floating point number in one FPAC by the floating point number in another FPAC.
FNEG	Negate	Inverts the sign bit of the FPAC.
FNOM	Normalize	Normalizes the floating point number in FPAC.
FNS	No Skip	The next sequential word is executed.
FPOP	Pop Floating Point State	Pops an 18-word floating point block off the user stack and alters the state of the floating point unit.
FPSH	Push Floating Point State	Pushes an 18-word floating point block onto the user stack.
FRH	Read High Word	Places the high-order 16 bits of an FPAC in AC0.
FSA	Skip Always	The next sequential instruction is skipped.
FSCAL	Scale	Shifts the mantissa of the floating point number in FPAC either right or left, depending upon the contents of bits 1-7 of AC0.
FSEQ	Skip On Zero	Skips the next sequential word if the Z flag of the FPSR is 1.
FSGE	Skip On Greater Than Or Equal To Zero	Skips the next sequential word if the N flag of the FPSR is 0.
FSGT	Skip On Greater Than Or Equal To Zero	Skips the next sequential word if both the Z and N flags of the FPSR are 0.
FSLE	Skip On Less Than Or Equal To Zero	Skips the next sequential word if either the Z flag or the N flag of the FPSR is 1.
FSLT	Skip On Less Than Zero	Skips the next sequential word if the N flag of the FPSR IS 1.
FSMS, FSMD	Subtract (memory from FPAC)	Subtracts the floating point number in memory from the floating point number in an FPAC.
FSND	Skip On No Zero Divide	Skips the next sequential word if the divide by zero (DVZ) flag of the FPSR is 0.

FLOATING POINT (Continued)

Mnem	Name	Function
FSNE	Skip On Non-Zero	Skips the next sequential word if the Z flag of the FPSR is 0.
FSNER	Skip On No Error	Skips the next sequential word if bits 1-4 of the FPSR are all 0.
FSNM	Skip On No Mantissa Overflow	Skips the next sequential word if the mantissa overflow (MOF) flag of the FPSR is 0.
FSNO	Skip On No Overflow	Skips the next sequential word if the overflow (OVF) flag of the FPSR is 0.
FSNOD	Skip On No Overflow And No Zero Divide	Skips the next sequential word if both the overflow (OVF) flag and the divide by zero (DVZ) flag of the FPSR are 0.
FSNU	Skip On No Underflow	Skips the next sequential word if the underflow (UNF) flag of the FPSR is 0.
FSNUD	Skip On No Underflow And No Zero Divide	Skips the next sequential word if both the underflow (UNF) flag and the divide by zero (DVZ) flag of the FPSR are 0.
FSNUO	Skip On No Underflow And No Overflow	Skips the next sequential word if both the underflow (UNF) flag and the overflow (OVF) flag of the FPSR are 0.
FSS, FSD	Subtract (FPAC from FPAC)	Subtracts the floating point number in one FPAC from the floating point number in another FPAC.
FSST	Store Floating Point Status	Moves the contents of the FPSR to two memory locations.
FSTS, FSTD	Store Floating Point	Stores the contents of a specified FPAC into memory.
FTD	Trap Disable	Sets the trap enable flag of the FPSR to 0.
FTE	Trap Enable	Sets the trap enable flag of the FPSR to 1.

ALC MANIPULATION

ALC Format

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

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

1	ACS	ACD	OP CODE	SH	C	#	SKIP
0	1 2	3 4	5 6 7	8 9	10 11	12	13 14 15

ALC Instructions

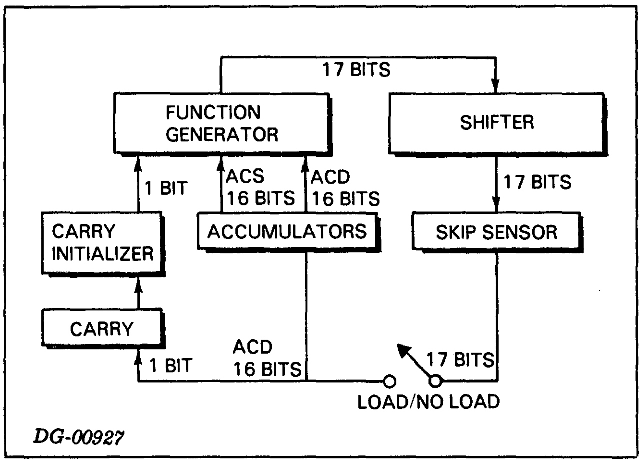
The ALC instructions are listed below.

ALC Instructions

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

ALC Instruction Execution

The ALC instructions use an Arithmetic Logic Unit (ALU) to process data. The logical organization of the ALU is illustrated below.



DG-00927

When an ALC instruction begins execution, it loads the contents of the carry bit and the contents of the accumulator(s) to be processed into the ALU. There are five distinct stages of ALU operation. We will discuss these stages separately.

Carry

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

SYMBOL	VALUE	OPERATION
[c] omitted	00	Leave Carry bit unchanged
[c]=Z	01	Initialize Carry bit to 0
[c]=O	10	Initialize Carry bit to 1
[c]=C	11	Complement the Carry bit

Function

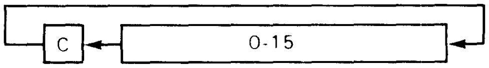
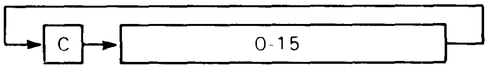
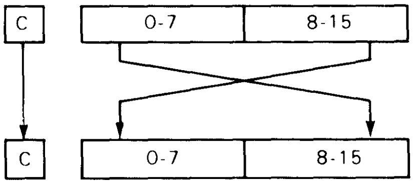
The ALU next evaluates the effect of the specific function (bits 5-7) upon the data. For the instructions *Move, Negate, AND,* and *Complement* the ALU performs the function on the data word(s) and saves the result. The value of the carry is as it was calculated above. For the instructions *Add, Add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

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

Shift Operations

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

SYMBOL	VALUE	OPERATION
[sh] omitted	00	Do not shift the result of the ALC operation
[sh]=L	01	Rotate left the 17-bit combination of Carry bit and ALC operation result
[sh]=R	10	Rotate right the 17-bit combination of Carry bit and ALC operation result
[sh]=S	11	Swap the two 8-bit halves of the ALC operation result without affecting Carry bit

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 

Skip Tests

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

SYMBOL	VALUE	OPERATION
[skip] omitted	000	No skip
[skip]=SKP	001	Skip unconditionally
[skip]=SZC	010	Skip if Carry bit is zero
[skip]=SNC	011	Skip if Carry bit is nonzero
[skip]=SZR	100	Skip if ALC result is zero
[skip]=SNR	101	Skip if ALC result is nonzero
[skip]=SEZ	110	Skip if either ALC result or Carry bit is zero
[skip]=SBN	111	Skip if both ALC result and Carry bit is nonzero

Load/No-Load

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

SYMBOL	VALUE	OPERATION
# omitted	0	Load the result of the shift operation into ACD
#	1	Do not load the ALC operation result into ACD; restore Carry bit to value it had before shifting

NOTE: *These instructions 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.*

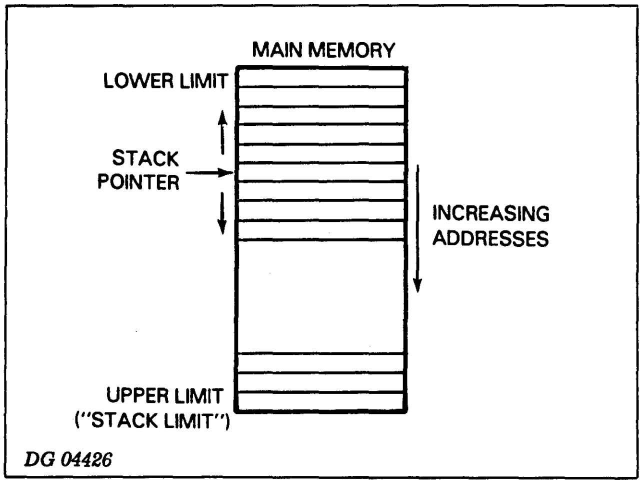
THE STACK

The stack is a series of consecutive locations in memory. In their simplest form, stack instructions add items in sequential order to the top of the stack and retrieve them in the reverse order. Several stack areas may be defined by the program, but only one stack may be in use at any time. The ECLIPSE C/150 uses the push-down stack concept to provide easily accessible temporary storage of data, variables, return addresses, etc.

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

The return block can take several forms, but it usually consists of five words: the contents of the four accumulators, the program counter or the frame pointer (see below), and the carry bit in bit 0 of the last word pushed.

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



To use the stack, define the upper and lower limits, then use the stack instructions to put items on *(push onto)* or remove items from *(pop off)* the top of the stack. It is not necessary to keep track of the location of the top of the stack. This is done automatically by the stack pointer. The updated value of the stack pointer is always stored in location 40_8.

The lower limit of the stack is determined by the initial value of the stack pointer, which is placed in location 40_8 when the stack is set up by the program. The upper limit is controlled by the value in location 42_8. This value is also chosen when the stack is set up, but it can be changed by the program if more stack area becomes necessary. Two other reserved locations are used to control the stack. Location 43_8 contains the address of the Stack Fault routine. Control is transferred to the Stack Fault routine when a stack underflow or overflow occurs (see Stack Protection, below). Location 41_8 contains the current value of the frame pointer, which is used as a reference pointer in the stack.

Stack Control Words

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

Stack Pointer

The stack pointer is the address of the current top of the stack. Its current value is always in location 40_8. A push operation increments the stack pointer by 1 and places the pushed word in the location addressed by the new value of the stack pointer. A pop operation takes the word addressed by the current value of the stack pointer, places it in a register and decrements the stack pointer by 1.

When the stack is set up, the value of the stack pointer is initially set to one less than the address of the first word in the stack. This determines the lower limit of the stack.

Stack Limit

The stack limit is the upper limit of the stack area. After each push operation, the stack pointer is compared with the stack limit. If the stack pointer is greater than the stack limit, an overflow condition exists. The stack limit is contained in location 42_8. For more information, see the next section on Stack Protection.

Stack Fault Address

If a stack overflow or underflow occurs, control is transferred to the Stack Fault routine. The address of this routine, which may be indirect, is contained in location 43_8.

Frame Pointer

The frame pointer differs from the stack pointer in that it is not changed by push or pop operations, and so its value is not incremented or decremented. This makes it a useful reference pointer when it is set to the same value as the stack pointer, because it then preserves the original value of the stack pointer.

The frame pointer is used by the *Save* and *Return* instructions to store and reset the value of the stack pointer when entering or leaving subroutines. The frame pointer can also be used to define the boundary between words placed in the stack by a calling routine and words placed by a called routine. Using the frame pointer as a reference, a routine can go back into the stack and retrieve variables left there by the preceding procedure.

The frame pointer is contained in location 41_8.

Stack Protection

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

Stack Overflow

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

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

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

Stack Underflow

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

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

Underflow protection can be disabled in two ways:

- Start the stack at a location greater than 401_8. A stack fault will not occur then unless the program underflows the stack and then continues to pop words off the stack until the stack pointer is less than 400_8. Note that this does not completely disable underflow protection - it is always possible to pop enough words off the stack to underflow it.
- Set bit 0 of both the stack pointer and the stack limit to 1. If this is done, all or a portion of the stack may reside in page zero (locations 0-377_8), or the stack may underflow into page zero, without interference from the stack underflow mechanism.

Stack Protection Faults

Stack Overflow Protection

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

- sets bit 0 of the stack pointer to 0;
- sets bit 0 of the stack limit to 1;
- pushes a return block onto the stack;
- executes a *jump indirect* to the stack fault address.

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

Stack Underflow Protection

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

- sets the stack pointer equal to the stack limit;
- sets bit 0 of the stack pointer to 0;
- sets bit 0 of the stack limit to 1;
- pushes a return block onto the stack;
- executes a *jump indirect* to the stack fault address.

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

Stack Fault Handler

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

Initializing the Stack Control Words

Initialize the stack control words before the first operation on the stack is performed. The rules for this are as follows:

Stack Pointer

- Initialize the stack pointer to the beginning address of the stack minus one.
- If stack underflow protection is desired, initialize the stack pointer to 400_8 and start the stack area at 401_8 .
- If stack underflow protection is not desired, start the stack at some location greater than 401_8 .
- If you want to have all or a portion of the stack area in page zero, or you want to disable underflow protection, set bit 0 of both the stack pointer and the stack limit to 1.

Stack Limit

- Initialize the stack limit to a value greater than the stack pointer.
- If stack overflow protection is desired, initialize the stack limit to the last address allocated for the stack minus at least 10.
- If stack overflow protection is not desired, initialize the stack limit to 77777_8.
- If you want to have all or a portion of the stack area in page zero, set bit 0 of both the stack pointer and the stack limit to 1.

Stack Fault Address

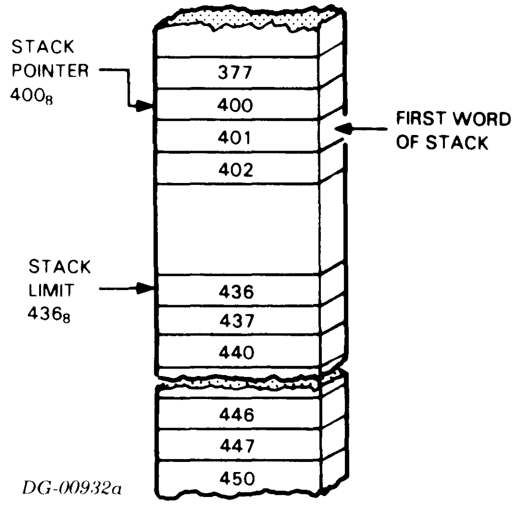
Initialize the stack fault address to the address of the routine that is to receive control in the event of a stack overflow or underflow. Bit 0 may be set to 1 to indicate an indirect address.

Frame Pointer

It is meaningless to attempt initialization of the frame pointer until it is actually used. The frame pointer will have no meaning until the first use of the *Save* instruction.

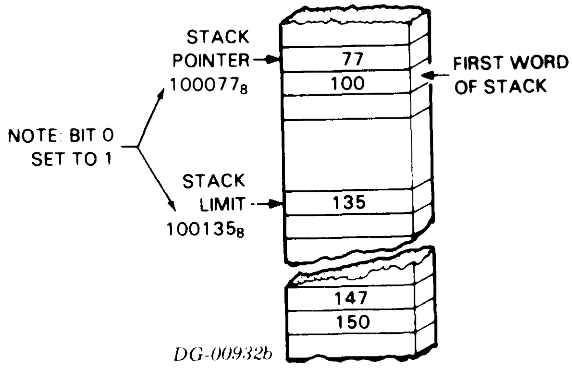
Examples

Stack area 50_8 words with underflow protection:



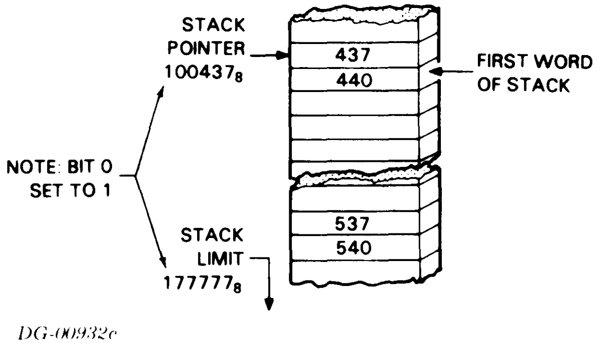
DG-00932a

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



DG-00932b

Stack area 100_8 words, no protection:



DG-00932c

The first of the preceding stack arrangements could be set up using the following assembly language instructions:

```
.TITL   STACK

.EXTN   STH                 ;Declare STH external

.LOC    401                 ;Go to location 401
.BLK    50                  ;Allocate 50 (octal) words

.LOC    40                  ;Go to stack control words
 400                        ;Stack pointer
 400                        ;Frame pointer
 436                        ;Stack limit
 STKHR                      ;Address of stack fault
                            ; handler
.END
```

Stack Instructions

The instructions that affect the stack are listed below.

STACK INSTRUCTIONS

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

PROGRAM EXECUTION

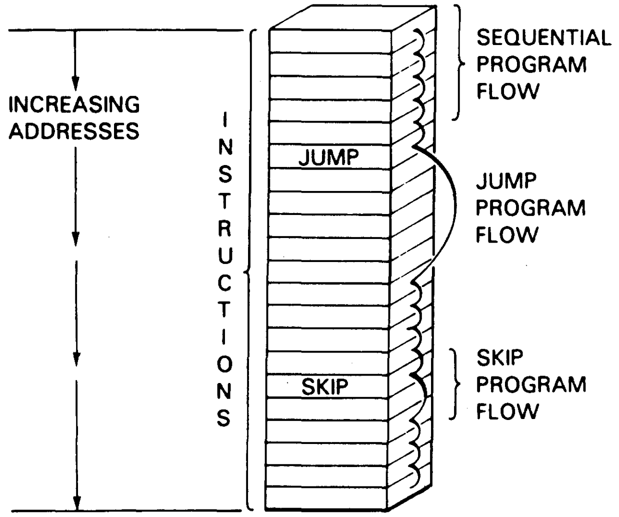
Sequential Operation

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

Program Flow Alteration

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

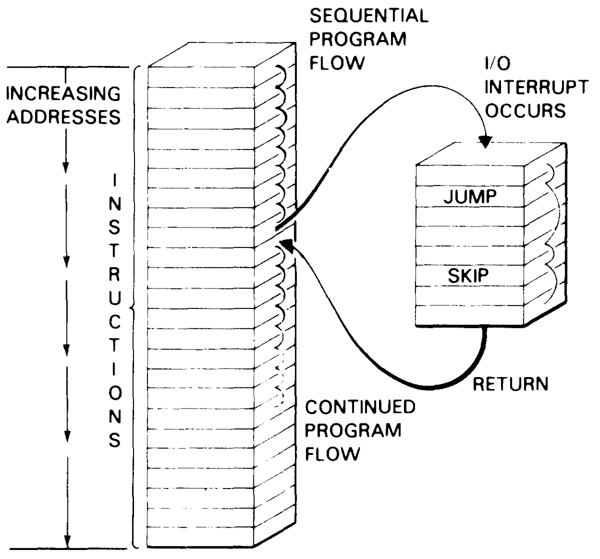
> NOTE: *Do not use a conditional skip immediately before a 2-word instruction. The conditional instruction causes a 1-word skip, which results in an attempt to execute the second word of the instruction as a 1-word instruction.*



DG-00543

Program Flow Interruption

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



DG-00544

Program Flow Alteration Instructions

Program flow alteration and conditional instructions are shown in the following tables.

In the first table, several instructions have both short and long forms. The short form is 16 bits in length and can directly specify a memory address from 0 to 255 or can directly specify a small area in memory surrounding the present value of the program counter or an accumulator. Long form instructions are 32 bits in length; they can directly specify any address from 0 to 77777_8.

The second table summarizes the skip instructions that test condition codes in the floating point status register.

The third table summarizes the condition tests available for the **SKIP**_[t]_ instruction. (This instruction tests condition codes of a peripheral device, the power-fail monitor or the interrupt system.)

The fourth table summarizes *skip* options of the ALC instructions.

PROGRAM FLOW ALTERATION INSTRUCTIONS

Mnem	Name	Function
CLM	Compare To Limits	Compares a signed integer with two other numbers and skips if first integer is between the other two.
DSPA	Dispatch	Compares a signed integer with two other numbers and skips if first integer is not between the others; otherwise, uses the integer as an index into a table and places indexed value in the program counter.
DSZ, EDSZ	Decrement And Skip If Zero	Decrements the addressed word, then skips if the decremented value is zero.
ISZ, EISZ	Increment And Skip If Zero	Increments the addressed word, then skips if the incremented value is zero.
JMP, EJMP	Jump	Places an effective address in the program counter.
JSR, EJSR	Jump To Subroutine	Increments program counter and stores incremented value in AC3; then places a new address in the program counter.
POPJ	Pop PC And Jump	Pops the top word off the stack and places it in the program counter.
PSHJ	Push	Pushes the address of the next sequential instruction onto the stack and places a new address in the program counter.
RSTR	Restore	Returns control from I/O interrupt handlers that use the stack change facility of the **VCT** instruction.
RTN	Return	Returns control from a subroutine entered via *Save* instruction.
SGE	Skip If ACS Greater Than Or Equal To ACD	Compares two signed integers in two accumulators and skips if the first is greater than or equal to the second.

Program Flow Alteration Instructions (Cont'd)

Mnem	Name	Function
SGT	Skip If ACS Greater Than ACD	Compares two signed integers in accumulators; skips if first is greater than the second.
SKP*[t]*	I/O Skip	Skips if the I/O condition *t* is true.
SNB	Skip On Nonzero Bit	References a single bit in memory via bit pointer; skips if bit is 1.
SYC SVC	System Call	Pushes a return block onto the stack; places address of *System Call* handler in program counter.
SZB	Skip On Zero Bit	References a single bit in memory via bit pointer; skips if bit is 0.
SZBO	Skip On Zero Bit, Set To 1	References a single bit in memory via bit pointer; skips if bit is 0 and also sets the bit to 1.
VCT	Vector On Interrupting Device Code	Identifies highest priority interrupt; passes control through a table to a handler routine for device.
XOP XOP1	Extended Operation	Pushes a retum block onto the stack, indexes into the XOP table and transfers control to another procedure.
XCT	Execute	Executes contents of an accumulator as an instruction.

FLOATING POINT SKIP TESTS

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

I/O Skip Tests

SYMBOL	FUNCTION
[t]=BN	Tests Busy flag for nonzero
[t]=BZ	Tests Busy flag for zero
[t]=DN	Tests Done flag for nonzero
[t]=DZ	Tests Done flag for zero

ALC Skip tests

SYMBOL	FUNCTION
[skip] omitted	No skip
[skip]=SKP	Skip unconditionally
[skip]=SZC	Skip if Carry bit is zero
[skip]=SNC	Skip if Carry bit is nonzero
[skip]=SZR	Skip if ALC result is zero
[skip]=SNR	Skip if ALC result is nonzero
[skip]=SEZ	Skip if either ALC result or Carry bit is zero
[skip]=SBN	Skip if both ALC result and Carry bit is nonzero

EXTENDED OPERATION FEATURE

The extended operation feature (XOP) provides an efficient method of transferring control to and from procedures. It enables the user to transfer control to any one of 32 procedure entry points.

Extended Operation Instuctions

There are two extended operation instructions in the ECLIPSE C/150 instruction set.

EXTENDED OPERATION INSTRUCTIONS

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

MEMORY ALLOCATION AND PROTECTION

MAP Functions

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 ECLIPSE C/150 MAP unit provides the hardware necessary to control and use more than 64 Kbytes of physical memory. 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 purposes of this discussion, we define certain words and phrases:

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

Physical Address - The address used by the MAP to address the physical memory. The maximum size of the physical address space is 1,048,576 words (1M) and it is addressed by a 20-bit address.

Address Translation - The process of translating logical addresses into physical addresses.

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

Page - 1024 (2000_8) words in memory.

User Map - The set of memory address translation functions defined for a particular user.

Data Channel Map - The set of address translation functions defined by the user-specified map. These are defined for the memory references of a data channel used by a particular device.

Supervisor - The section of the operating system (software) which controls system functions such as the operation of the 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 in the ECLIPSE C/150 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 in the ECLIPSE C/150. *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).*

Unmapped Mode

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

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 in the ECLIPSE C/150 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, however, the fifth word will always be the logical address of the instruction following the instruction that caused the fault.

Load Effective Address Mode

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

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.

Initial Conditions

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

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

MAP Instructions

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

> NOTE: *MAP instructions can be executed in mapped mode if I/O protection and* Lef *mode are disabled for that user. When executed in mapped mode, the* Read Map Status, Initiate Page Check, *and* Page Check *instructions will return the desired information without changing the map. The* Map Single Cycle *instruction will disable the user map after the next memory reference. The remainder of the instructions will change the map while the map is enabled, with undesirable results for this user, another user, or the system as a whole.*
>
> *Enabling* Lef *mode only will convert all I/O instructions (including MAP instructions) to* Lef *instructions. The* Load Map *instruction, however, does not use the I/O format and therefore can still be executed. Enabling both* Lef *mode and I/O protection will prevent execution of the* Load Map *instruction.*

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

MAP INSTRUCTIONS

Mnem	Name	Function
DIA	Read Map Status	Reads the status of the current map.
DIC	Page Check	Provides the identity and some characteristics of the physical page corresponding to the logical page identified by the immediately preceding *Initiate Page Check* instruction.
DOA	Load Map Status	Defines the parameters of a new map.
DOB	Map Supervisor Page 31	Specifies the physical page corresponding to logical page 31 of the supervisor's address space.
DOC	Initiate Page Check	Identifies a logical page.
LMP	Load Map	Loads successive words from memory into the MAP where they are used to define a user or data channel map.
NIOP	Map Single Cycle	Maps one memory reference using the last user map.

INPUT/OUTPUT

This section describes the Input/Output (I/O) of the ECLIPSE C/150. We first discuss the general operation of the system, then interrupts and the *Vector* instruction.

The ECLIPSE C/150 has a 6-bit device selection network, corresponding to bits 10-15 in the I/O instruction format. The devices are connected to this network in such a way that each device will only respond to commands sent with its own device code. With a 6-bit device code, 64 separate devices can be individually controlled. Some of these device codes are reserved for the CPU and certain processor options, but the remaining are available for referencing I/O devices. The assembler recognizes mnemonics for those devices assigned a code by Data General. A complete list of these is provided in Appendix A of this manual.

See *Programmer's Reference Manual - Peripherals (DGC No. 015-00021)* for details about programming specific devices in the I/O system.

Busy and Done Flags

I/O devices are controlled by manipulating their Busy and Done flags (but note that data channel devices require several programmed I/O instructions to be properly set up before they can be started with the flags). You can change the value of these flags using optional flag control command mnemonics appended to the instruction. When Busy and Done are both 0, the device is idle and cannot perform any operations. To start a device, the program must set Busy to 1 and Done to 0. When the device has finished its operation and is ready to start another, it sets Busy to 0 and Done to 1.

Programmed I/O

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

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

Data Channel I/O

Data channel I/O permits data to be transferred in blocks of words, with program control necessary only at the start of the operation. The CPU stops during each word transfer but the transfer is made directly to or from memory, so no additional steps are required. Data channel I/O is a very efficient method of transferring large blocks of data between memory and a fast I/O device. When single words or bytes are needed, however, programmed I/O is generally faster.

The maximum transfer rate for data channel I/O is as follows:

- Input: One word every 800 ns, or 1,250,000 words per second,
- Output: One word every 1400 ns, or 715,000 words per second.

At these rates, the CPU is effectively stopped. At lower rates, however, processing continues while data is being transferred.

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

When a data channel device is ready to send or receive data, it issues a data channel request to the processor. 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 data channel between the device and memory without specific action by the program.

All requests are honored according to the relative position of the requesting devices on the I/O bus. The device requesting data channel service which is physically closest on the bus is serviced first, the next closest device next, 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.

After handling all data channel requests, the processor then handles all outstanding I/O interrupt requests. Only then does program execution continue.

For more information on the data channel, see *Programmer's Reference Manual - Peripherals (DGC No. 015-000021)* and *User's Manual - Interface Designer's Reference (DGC No. 015-000031).*

I/O Interrupts

The I/O interrupt system in the ECLIPSE C/150 provides a convenient method of handling programmed I/O with a minimum of overhead. Instead of polling each I/O device repeatedly to find out when it is ready to transmit or receive data, the interrupt system permits the program to ignore the I/O devices completely until one requires service. At that time, the device requests an interrupt. As soon as the processor is at an interruptable point in its processing, and has finished servicing data channel requests, it services the interrupt.

Interrupt System Definitions

Interrupt request line- - Common connection between all I/O devices and the computer. An I/O device places a request on the interrupt request line at the same time that it sets Busy to 0 and Done to 1, i.e., when it has finished a task and is ready to send or receive data. No information is placed on the line which permits the program to determine which device is requesting an interrupt. This must be done separately.

Interrupt On flag- - Flag in the CPU which 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 does not look at the interrupt request line at all, and therefore does not respond to any interrupts.

Priority mask- - Set of bits in the I/O devices that control the priority interrupt system. Each I/O device is connected to one of 16 bits in the priority mask. Some bits are connected to more than one I/O device. When a bit is set to 1, the devices connected to it cannot place a request on the interrupt request line, although they can set their Busy flags to 0 and their Done flags to 1. Since the mask can be changed by the program, different devices can be inhibited at different times to conform to the needs of a priority system.

Base level- - The state of a program when no I/O devices are inhibited (all mask bits are 0) and no interrupt processing is in progress. This is the environment in which user program execution takes place.

Nonbase level- - Any system state in which some I/O devices are inhibited and/or interrupt processing is in progress. Interrupt handlers operate at non-base level.

In the next section we will discuss interrupts. First we will discuss interrupts without a priority system, and then we will consider a priority interrupt system.

Processing an Interrupt Without a Priority System

When an I/O device completes its operation and is ready to send or receive more data, it sets its Busy flag to 0 and its Done flag to 1. Since its priority bit is 0, it also places a request on the interrupt request line. If the Interrupt On flag is 1 when the processor is next interruptable, the interrupt will be serviced.

When servicing an interrupt, the CPU first sets the Interrupt On flag to 0 so that no devices can interrupt the first part of the interrupt service routine. If a user map is enabled, it is disabled. The CPU then places the contents of the updated program counter into physical memory location 0 and jumps indirect via location 1, where it expects to find the address (direct or indirect) of the interrupt service routine.

The interrupt service routine (supplied by the user) must save any accumulators that will be used, save the carry bit if it will be used, determine which device requested the interrupt, and then service that device as necessary.

The service routine can identify the interrupting device by using *I/O skip* instructions, or the *Interrupt acknowledge* instruction. Or it can save the return information and identify the interrupting device with one instruction by using 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* instruction and uses the code returned as an index into a table of addresses. These addresses are the beginnings of the various device service routines.

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

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

Priority Interrupt System

The need for a priority interrupt system can be illustrated as follows:

If the Interrupt On flag remains 0 throughout the interrupt service routine, the CPU cannot be interrupted while an I/O device is being serviced. All other devices, therefore, must wait until the first device is finished. If the Interrupt On flag is returned to 1 after the initial portion of the service routine, any I/O device can interrupt the servicing of any other I/O device. While this might be reasonable for some devices, it is not for others. It is therefore desirable to have a system of interrupt priorities which will permit some devices to interrupt certain others without disrupting the orderly processing of data.

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

The ECLIPSE C/150 has the hardware and instructions for a more flexible and efficient priority system, with up to sixteen levels of priority interrupts. The interrupt service routine has full control of this system, and can change the priorities of various devices as necessary.

Setting Up a Priority System

To set up a system of priorities, place a *Mask Out* instruction in the interrupt service routine for each device. This instruction changes the priority mask, thus controlling which devices can interrupt. All those devices which should not interrupt the device being serviced are masked out (prevented from requesting an interrupt) if their mask bits are 1. In addition, all pending interrupt requests from devices controlled by that bit are disabled. The other mask bits, corresponding to the devices which can interrupt, are set to 0.

If this is done in each interrupt service routine, then the mask will always mask out those devices which should not interrupt the device presently being serviced. This is a dynamic process, changing each time a different device is serviced, resulting in a system of priorities. The device with the highest priority will be able to interrupt all other devices, and the device with the lowest priority will be interruptible by all other devices.

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

A multiple priority level interrupt handler must be interruptable without damage. Usually this is not true for the initial portions of the interrupt handler, so the Interrupt On flag is initially set to 0. The interrupt handler must first save return information after receiving control. This information must 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 of the previous level.

Next, the correct service routine must be chosen. This routine must save the current priority mask and establish a new one. Once this is all completed, the *Interrupt Enable* instruction can be used to set the Interrupt On flag to 1, enabling those devices not restricted by the priority mask to interrupt if necessary.

After servicing the interrupt, the interrupt service routine should:

- disable the interrupt system,
- reset the priority mask to the condition it was in when the routine was entered,
- restore the accumulators and the carry bit,
- enable the interrupt system,
- return control to the interrupted program.

Stack Changes

The interrupt handler usually requires use of a stack. Rather than work with the user stack, you can define a new stack which is reserved for use by the interrupt handler. This overcomes the following problems:

- There is no guarantee that a user stack will always be defined,
- The user stack pointer could be just below the stack limit. The interrupt handler would then overflow the user stack.

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

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

The *Vector* instruction handles all these stack changes by using different modes in different situations. The next section will discuss the use of this instruction.

Using the Vector Instruction

The *Vector on interrupting device code* instruction can simplify the design of an interrupt handler by doing many of the required steps in one instruction. It can also perform different levels of tasks as needed within the interrupt handler.

The *Vector* instruction has five different modes that can be used in different circumstances. The simplest of these is scarcely more complex than the *Interrupt acknowledge* instruction. It does not save any information on the state of the computer at the interrupt, and takes very little time. The most complex mode, on the other hand:

- saves considerable information on the state of the machine,
- stores the user stack parameters,
- creates a new stack,
- resets the priority mask,

and, of course, takes much longer.

When choosing which mode to use, you must weigh the importance of saving the state of the computer, having a separate vector stack, and changing the priority mask, against the time used for each interrupt. Note that you are not committed to one mode throughout the interrupt handler. It is possible to use different *Vector* instruction modes at different times to serve different needs. An example at the end of this section illustrates this.

Mode A - is used when a device requires immediate interrupt service. This would be the case for unbuffered devices with very short latency times, or for real time processes that require immediate access. The price you pay for fast reaction time is that nothing is saved to make the return from the interrupt easier.

Modes B through E - all create a priority structure which permits some interrupting devices to interrupt the service of certain others. This takes longer than mode A service, but permits devices which need immediate service to get it even if a slower device is already being serviced.

Modes D and E - both initiate a new stack. You should use them only when operating at base level (no interrupt processing in progress) since they set up a new vector stack for use by the interrupt handler and store the (old) user stack parameters in it. Once this new stack has been set up, there is no reason to try to set it up again if a new interrupt occurs before the old one was finished. Mode E also pushes a return block onto the stack to make return to the first interrupt handler easier.

Modes B and C - do not initiate a new stack, and are therefore appropriate to use when operating at non-base level (that is, when a device interrupts the interrupt processing of another device). Mode C also pushes a new return block onto the stack. Some of the S/400 I/O instructions have special mnemonics which can be used in place of the standard mnemonics. Note that the mnemonics for controlling the state of flags cannot be appended to these special instruction mnemonics.

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

DOB*f ac*,CPU

instead of the special mnemonic:

MSKO *ac*

The special mnemonic sets bits 8 and 9 to 00.

I/O INSTRUCTIONS

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

POWER FAIL/AUTO-RESTART

When power is turned off and then on again, core memory is unaltered, but the contents of semiconductor memory are lost. The state of the accumulators, the program counter, and the various flags in the CPU and SC memory then are indeterminate. The power fail facility provides a *fail-soft* capability in the event of unexpected power loss.

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

When power is restored, the action taken by the automatic restart portion of the power fail facility 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 the instruction contained in physical location 0, thereby transferring control to the restart procedure.

The contents of semiconductor memory are lost under a power failure. Therefore, the auto restart facility should not attempt to restart the system, even with the power switch in the LOCK position, if the host contains semiconductor memory. This can be controlled by proper positioning of jumpers on the power fail facility. The local memory of any optional IOP and/or DCU/50 in the system is semiconductor so the restart facility must reload those memories before restarting the processors.

POWER FAIL

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

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

POWER FAIL INSTRUCTIONS

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

MEMORY ERROR CHECKING

Error Checking And Correction

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

Each ERCC memory word is 21 bits long. These 21 bits consist of 16 data bits followed by 5 ERCC check bits. Each time the CPU writes data into a location, a hardware encoder constructs the check field bits from the 16 data bits. When the CPU reads a memory location, the encoder checks the ERCC bits read from memory. If the 21 bits do not generate an error code when read, the ERCC facility passes the 16 data bits on to the CPU. If the 21 bits generate an error code, a single bit error has occurred. The memory pauses while the ERCC facility corrects the single bit in error and rewrites the entire corrected word back into the memory location. The ERCC facility then passes the data on to the CPU and 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.

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

ERCC Instructions

The operation of the ERCC facility is governed by one I/O instruction. Two other instructions are used to interrogate ERCC after it has detected and corrected an error. ERCC contains a Done flag which is set to 1 after an error has been detected and initiates an interrupt request. A fourth instruction is used to set this flag to 0. The ERCC facility has no Busy flag and no mask bit in the priority mask. The device code for the ERCC facility is 2. The assembler recognizes the mnemonic ERCC for this device code.

All the ERCC instructions with the exception of the *Clear ERCC interrupt request* use an accumulator, which is specified when coding the instruction, to receive the data or contain the control information.

ERCC INSTRUCTIONS

Mnem	Name	Function
DOA	Enable ERCC	Enables the ERCC facility according to the setting of bits 14-15 of the specified accumulator.
DIA	Read Memory Fault Address	Returns the low-order bits of the memory location which has produced an error.
DIB	Read Memory Fault Code	Returns a 5-bit error code which tells which bit was in error. Also returns the high-order bits of the memory fault address.
NIOS	Clear ERCC Interrupt Request	Sets the ERCC Done flag to 0; clears an interrupt request if one was pending.

RESERVED STORAGE LOCATIONS

The following location are reserved storage locations in the ECLIPSE C/150. These locations are used for specific functions by the CPU and should not be used for other functions.

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

The following locations are in unmapped logical address space:

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

The following locations may be in unmapped logical address space or in Map A or Map B logical address space. They are usually placed in unmapped logical address space:

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

The following locations are usually in Map A or Map B logical address space, but they may be in unmapped addressed space too:

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

Chapter III
ECLIPSE C/150 INSTRUCTIONS

This chapter lists all the instructions for the machine *except* those I/O instructions intended for a specific device such as the MAP, the BMC, and special CPU instructions. We have arranged the instructions in alphabetical order according to mnemonics as recognized by the assembler.

For each instruction we include:

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

Some instructions can only be executed by the host processor, while others can also be executed by the I/O processor and/or the Data Control Unit. A label with each instruction indicates which processors can execute that instruction.

CODING AIDS

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

[] *[]* Square brackets indicate that the enclosed symbol (e.g., *[,skip])* is an optional operand or mnemonic. Code it only if you want to specify the option.

BOLD Code operands or mnemonics printed in boldface exactly as shown. For example, code the mnemonic for the *Move* instruction: **MOV**.

italic For each operand or mnemonic in italics, replace the item with a number or symbol that provides the assembler value you need for that item (e.g., the proper accumulator number, an address, etc.).

We use the following abbreviations throughout this chapter:

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

Add Complement

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

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

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

Initializes the carry bit to the specified value, adds the logical complement of the unsigned, 16-bit number in ACS to the unsigned, 16-bit number in ACD, and places the result in the shifter. If the addition produces a carry of 1 out of the high-order bit, the carry bit is complemented. The instruction then performs the specified shift operation, and loads the result of the shift into ACD if the no-load bit is 0. If the skip condition is true, the next sequential word is skipped.

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

Add

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

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

Performs unsigned integer addition and complements the carry bit if appropriate.

Initializes the carry bit to the specified value, adds the unsigned, 16-bit number in ACS to the unsigned, 16-bit number in ACD, and places the result in the shifter. If the addition produces a carry of 1 out of the high-order bit, the carry bit is complemented. The instruction then performs the specified shift operation and places the result of the shift in ACD if the no-load bit is 0. If the skip condition is true, the next sequential word is skipped.

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

Extended Add Immediate

ADDI *i,ac*

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

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

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

Treats the contents of the immediate field as a signed, 16-bit, two's complement number and adds it to the signed, 16-bit, two's complement number contained in the specified accumulator, placing the result in the same accumulator. The Carry bit remains unchanged.

Add Immediate

ADI *n,ac*

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

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

Adds the contents of the immedi te field *N*, plus 1, to the unsigned, 16-bit number contained in the specified accumulator, placing the result in the same accumulator. 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, you should code the exact value that you wish to add.*

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

AND With Complemented Source

ANC *acs,acd*

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

Forms the logical AND of the logical complement of the contents of ACS and the contents of ACD and places the result in ACD. The instruction sets a bit position in the result to 1 if the corresponding bit positions in ACS and ACD contain a 0 and a 1, respectively. The contents of ACS remain unchanged.

AND

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

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

Forms the logical AND of the contents of two accumulators.

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

AND Immediate

ANDI *i,ac*

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

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

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

Block Add and Move

BAM

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

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

Moves words sequentially from one memory location to another, treating them as unsigned, 16-bit integers. After fetching a word from the source location, the instruction adds the unsigned, 16-bit integer in AC0 to it. If the addition produces a carry of 1 out of the high-order bit, no indication is given.

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

The unsigned, 16-bit number in AC1 is equal to the number of words moved. This number must be greater than 0 and less than or equal to 32,768. If the number in AC1 is outside these bounds, no data is moved and the contents of the accumulators remain unchanged.

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

For each word moved, the count in AC1 is decremented by one and the source and destination addresses in AC2 and AC3 are incremented by one. Upon completion of the instruction, AC1 contains zeroes, and AC2 and AC3 point to the word following the last word in their respective fields. The contents of AC0 remain unchanged.

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

> NOTE: *Because of the potentially long time that may be required to perform this instruction it 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 interrupted instruction. Because the addresses and the word count are updated after every word stored, any interrupt service routine that returns control to the interrupted program via the address stored in memory location 0 will correctly restart the* Block Add and Move *instruction.*

When updating the source and destination addresses, the *Block Add And Move* instruction forces bit 0 of the result to 0. This ensures that upon return from an interrupt, the *Block Add And Move* instruction will not try to resolve an indirect address in either AC2 or AC3.

Block Move

BLM

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

Moves memory words from one location to another.

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

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

Set Bit To One

BTO *acs,acd*

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

Sets the specified bit to 1.

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

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

Set Bit To Zero

BTZ *acs,acd*

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

Sets the addressed bit to 0.

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

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

Compare To Limits

CLM *acs,acd*

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

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

Compares the signed, two's complement integer in ACS to two signed, two's complement limit values, *L* and *H*. If the number in ACS is greater than or equal to L and less than or equal to H, the next sequential word is skipped. If the number in ACS is less than L or greater than H, the next sequential word is executed.

If ACS and ACD are specified as different accumulators, the address of the limit value L is contained in bits 1-15 of ACD. The limit value H is contained in the word following L. Bit 0 of ACD is ignored.

If ACS and ACD are specified as the same accumulator, then the integer to be compared must be in that AC and the limit values L and H must be in the two words following the instruction. L is the first word and H is the second word. The next sequential word is the third word following the instruction.

Character Compare

CMP

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

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

The instruction compares the strings one byte at a time. Each byte is treated as an unsigned 8-bit binary quantity in the range ($0-255_{10}$). If two bytes are not equal, the string whose byte has the smaller numerical value is, by definition, the *(lower valued)* string. Both strings remain unchanged. The four accumulators contain parameters passed to the instruction. Two accumulators specify the starting address, the number of bytes, and the direction of processing (ascending or descending addressed) for each string.

AC0 specifies the length and direction of comparison for string 2. If the string is to be compared from its lowest memory location to the highest, ACO contains the unsigned value of the number of bytes in string 2. If the string is to be compared from its highest memory location to the lowest, AC0 contains the two's complement of the number of bytes in string 2.

AC1 specifies the length and direction of comparison for string 1. If the string is to be compared from its lowest memory location to the highest, AC0 contains the unsigned value of the number of bytes in string 1. If the string is to be compared from its highest memory location to the lowest, AC1 contains the two's complement of the number of bytes in string 1.

AC2 contains a byte pointer to the first byte compared in string 2. When the string is compared in ascending order, AC2 points to the lowest byte. When the string is compared in descending order, AC2 points to the highest byte.

AC3 contains a byte pointer to the first byte compared in string 1. When the string is compared in ascending order, AC3 points to the lowest byte. When the string is compared in descending order, AC3 points to the highest byte.

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

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

Upon completion, AC0 contains the number of bytes left to compare in string 2. AC1 contains the return code as shown in the table above. AC2 contains a byte pointer either to the failing byte in string 2 (if an inequality was found), or to the byte following string 2 (if string 2 was exhausted). AC3 contains a byte pointer either to the failing byte in string 1 (if an inequality was found), or to the byte following string 1 (if string 1 was exhausted). If the length of both string 1 and string 2 was zero, the instruction returns 0 in AC1. If the two strings are of unequal length, the instruction *fakes* space characters $<040_8>$ in place of bytes from the exhausted string, and continues the comparison.

Character Move Until True

CMT

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

Under control of the four accumulators, moves a string of bytes from one area of memory to another until either a table-specified delimiter character is moved or the source string is exhausted.

The instruction copies the string one byte at a time. Before it moves a byte, the instruction uses that byte's value to determine if it is a delimiter. It treats the byte as an unsigned 8-bit binary integer (in the range 0-255_{10}) and uses it as a bit index into a 256-bit delimiter table. If the indexed bit in the delimiter table is zero, the byte pending is not a delimiter, and the instruction copies it from the source string to the destination string. If the indexed bit in the delimiter table is 1, the byte pending is a delimiter; the instruction does not copy it, and the instruction terminates.

The instruction processes both strings in the same direction, either from lowest memory locations to highest (*ascending order*), or from highest memory locations to lowest (*descending order*). Processing continues until there is a delimiter or the source string is exhausted. The four accumulators contain parameters passed to the instruction.

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

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

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

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

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

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

> NOTE: *If AC1 contains the number 0 at the beginning of this instruction, no bytes are fetched and none are stored. The instruction becomes a No-Op.*

Character Move

CMV

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

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

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

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

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

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

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

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

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

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

If the source field is shorter than the destination field, the instruction pads the destination field with space characters <040_8>. If the source field is longer than the destination field, the instruction terminates when the destination field is filled and returns the value 1 in the Carry bit, otherwise the instruction returns the value 0 in the Carry bit.

Character Translate

CTR

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

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

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

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

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

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

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

AC3 contains a byte pointer to the first byte in string 3.

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

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

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

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

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

Count Bits

COB *acs,acd*

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

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

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

Complement

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

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

Forms the logical complement of the contents of an accumulator.

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

Decimal Add

DAD *acs,acd*

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

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

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

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

Example:

Assume that bits 12-15 of AC2 contain 9; bits 12-15 of AC3 contain 7; and the carry bit is 0. After the instruction DAD 2,3 is executed, AC2 remains the same; bits 12-15 of AC3 contain 6; and the carry bit is 1, indicating a decimal carry from this *Decimal Add.*

	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

Double Hex Shift Left

DHXL *n,ac*

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

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

> NOTE: *If AC is specified as AC3, then AC+1 is AC0.*
>
> *The assembler takes the coded value of* n *and subtracts one from it before placing it in the immediate field. Therefore, the programmer should code the exact number of hex digits that he wishes to shift.*
>
> *If N is equal to 3, the contents of AC+1 are placed in AC and AC+1 is filled with zeroes.*

Double Hex Shift Right

DHXR *n,ac*

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

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

> NOTE: *If AC is specified as AC3, then AC+1 is AC0.*
>
> *The assembler takes the coded value of* n *and subtracts one from it before placing it in the immediate field. Therefore, the programmer should code the exact number of hex digits that he wishes to shift.*
>
> *If N is equal to 3, the contents of AC are placed in AC+1 and AC is filled with zeroes.*

Unsigned Divide

DIV

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

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

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

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

Signed Divide

DIVS

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

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

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

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

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

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

Double Logical Shift

DLSH *acs,acd*

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

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

AC3+1 is AC0. The number of bits shifted is equal to the magnitude of the number in bits 8-15 of ACS. Bits shifted out are lost, and the vacated bit positions are filled with zeroes. 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 31_{10}, all bits of ACD are set to 0. The Carry bit and the contents of ACS remain unchanged.*

Decimal Subtract

DSB *acs,acd*

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

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

Subtracts the unsigned decimal digit contained in ACS bits 12-15 from the unsigned decimal digit contained in ACD bits 12-15. Subtracts the complement of the carry bit from this result. Places the decimal units' position of the final result in ACD bits 12-15 and the complement of the decimal borrow in the carry bit. In other words, if the final result is negative, the instruction indicates a borrow and sets the carry bit to 0. If the final result is positive, the instruction indicates no borrow and sets the carry bit to 1. The contents of ACS and bits 0-11 of ACD remain unchanged.

Example:

Assume that bits 12-15 of AC2 contain 9; bits 12-15 of AC3 contain 7; and the carry bit is 0. After the instruction 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

Dispatch

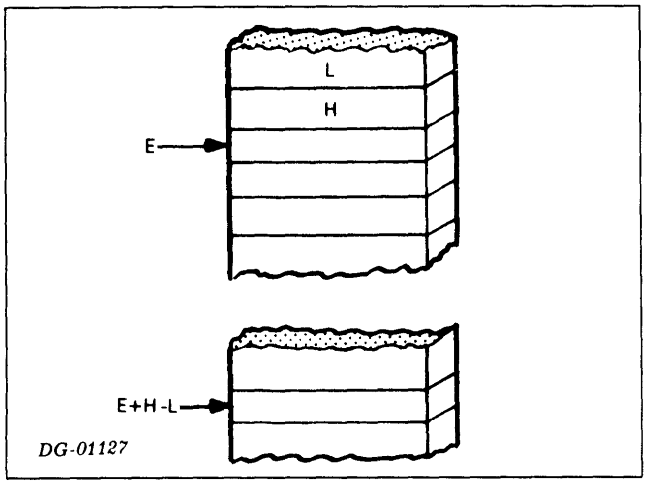
DSPA *ac,[@]displacement[,index]*

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

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

Conditionally transfers control to an address selected from a table.

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



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

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

Decrement And Skip If Zero

DSZ *[@]displacement[,index]*

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

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

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

Edit

EDIT

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

Converts a decimal number from either packed or unpacked form to a string of bytes under the control of an edit 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. The instruction also performs operations on alphanumeric data if data type 4 is specified.

The instruction maintains two flags and three indicators or pointers.

The flags are the significance Trigger (T) and the Sign flag (S). T is set to 1 when the first 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 to reflect the sign of the number being processed. If the number is positive, S is set to 0. If the number is negative, S is set to 1.

The three indicators are the Source Indicator (SI), the Destination Indicator (DI), and the op-code Pointer (P). Each is 16 bits wide and contains a byte pointer to the *current* byte in each respective area. At the beginning of an *Edit* instruction, SI is set to the value contained in AC3. DI is set to the value contained in AC2, and P is set to the value contained in AC0. Also at this time the sign of the source number is checked for validity.

The sub-program is made up of 8-bit op-codes followed by one or more 8-bit operands. P, a byte pointer, acts as the *program counter* for the *Edit* sub-program. The sub-program proceeds sequentially until a branching operation occurs - much the same way programs are processed. Unless instructed to do otherwise, the *Edit* instruction updates P after each operation to point to the next sequential op-code. The instruction continues to process 8-bit op-codes until directed to stop by the DEND op-code.

The sub-program can test and modify S and T, as well as modify SI, DI and P.

Upon entry to EDIT AC0 is a byte pointer to the first op-code of the *Edit* sub-program.

AC1 is the data-type indicator describing the number to be processed.

AC2 is a byte pointer to the the first byte of the destination field.

AC3 is a byte pointer to the first byte of the source field.

The fields may overlap in any way. However the instruction processes characters one at a time, so unusual side effects may be produced by certain types of overlap.

Upon successful termination, the carry bit contains the significance Trigger; AC0 contains a byte pointer (P) to the next op-code to be processed; AC1 is undefined; AC2 contains a byte pointer (DI) to the next destination byte; and AC3 contains a byte pointer (SI) to the next source byte.

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

Some op-codes perform movement of characters from one string to another. For those op-codes which move numeric data, special actions may be performed. For those which move non-numeric data, characters are copied exactly to the destination.

The Edit *instruction places information on the stack. Therefore, the stack must be set up and have at least 9 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 Edit *instruction assumes it is restarting from an interrupt; therefore do not allow this to occur under any other circumstances.*

In the description of some of the *Edit* op-codes we use the symbol *j* to denote how many characters a certain operation should process. When the high order bit of *j* is 1, *j* has a different meaning, it is a pointer into the stack to a word that denotes the number of characters the instruction should process. So, in those cases where the high order bit of *j* is 1, the instructions interpret *j* as an 8-bit two's complement number pointing into the stack. The number on the stack is at address:

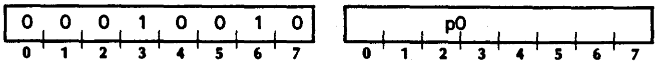
stack pointer + 1 + *j*.

The operation uses the number at this address as a character count instead of *j*.

An *Edit* operation that processes numeric data (e.g., DMVN) skips a leading or trailing sign code it encounters; similarly, such an operation converts a high-order or low-order sign to its correct numeric equivalent.

Add To DI

DADI *p0*



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

Add To SI

DASI *p0*

0	1	2	3	4	5	6	7
0	0	0	1	0	0	1	1

0	1	2	3	4	5	6	7
p0							

Adds the 8-bit two's complement integer specified by *p0* to the Source Indicator (SI).

Add To P Depending On T

DAPT *p0*

0	1	2	3	4	5	6	7
0	0	0	0	1	0	1	1

0	1	2	3	4	5	6	7
p0							

If T is one, the instruction adds the 8-bit two's complement integer specified by *p0* 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*

0	1	2	3	4	5	6	7
0	0	0	0	1	1	1	1

0	1	2	3	4	5	6	7
p0							

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

Add To P

DAPU *p0*

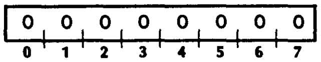
0	1	2	3	4	5	6	7
0	0	0	1	0	1	1	1

0	1	2	3	4	5	6	7
p0							

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

End Edit

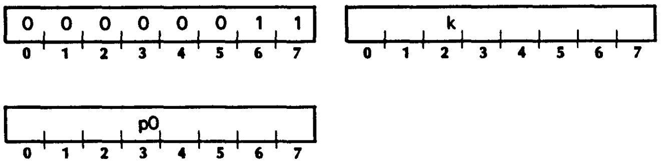
DEND



Terminates the EDIT sub-program.

Decrement and Jump If Non-Zero

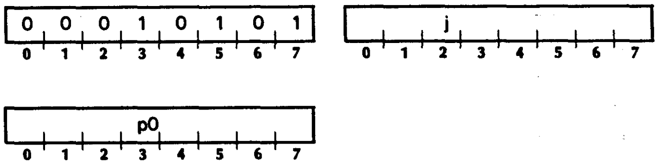
DDTK *k,p0*



Decrements a word in the stack by one. If the decremented value of the word is non-zero, the instruction adds the 8-bit two's complement integer specifed by *p0* 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 Character J Times

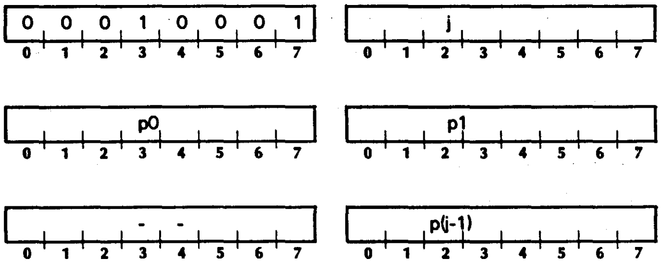
DIMC *j,p0*



Inserts the character specified by *p0* into the destination field a number of times equal to *j* beginning at the position specified by DI. Increases DI by *j*.

Insert Characters Immediate

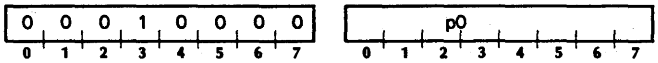
DICI *j,p0,p1,...,p(j-1)*



Inserts *j* characters from the op-code stream into the destination field beginning at the position specified by DI. Increases P by (*j* +2), and increases DI by *j*.

Insert Character Once

DINC *p0*



Inserts the character specified by *p0* in the destination field at the position specified by DI. Increments DI by 1.

Insert Sign

DINS *p0,p1*

0	1	2	3	4	5	6	7
0	0	0	0	1	1	1	0

0	1	2	3	4	5	6	7
p0							

0	1	2	3	4	5	6	7
p1							

If the Sign flag (S) is 0, the instruction inserts the character specified by *p0* in the destination field at the position specified by DI. If S is 1, the instruction inserts the character specified by *p1* in the destination field at the position specified by DI. Increments DI by one.

Insert Character Suppress

DINT *p0,p1*

0	1	2	3	4	5	6	7
0	0	0	0	1	0	1	0

0	1	2	3	4	5	6	7
p0							

0	1	2	3	4	5	6	7
p1							

If the significance Trigger (T) is 0, the instruction inserts the character specifed by *p0* in the destination field at the position specified by DI. If T is 1, the instruction inserts the character specified by *p1* in the destination field at the position specified by DI. Increments DI by one.

Move Alphabetics

DMVA *j*

0	1	2	3	4	5	6	7
0	0	0	0	1	1	0	1

0	1	2	3	4	5	6	7
j							

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

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

Move Characters

DMVC *j*

0	1	2	3	4	5	6	7
0	0	0	0	1	1	0	0

0	1	2	3	4	5	6	7
j							

Increments SI if the source data type is 3 and $j>0$. The instruction then moves *j* characters from the source field beginning at the position specified by SI to the destination field beginning at the position specified by DI. Increases both SI and DI by *j*. Sets T to 1.

Initiates a commercial fault if the attribute specifier word indicates that the source is data type 5 (packed). Performs no validation of the characters.

Move Float

DMVF *j,p0,p1,p2*

0	1	2	3	4	5	6	7
0	0	0	1	0	1	0	0

0	1	2	3	4	5	6	7
j							

0	1	2	3	4	5	6	7
p0							

0	1	2	3	4	5	6	7
p1							

0	1	2	3	4	5	6	7
p2							

If the source data type is 3, j >0, and SI points to the sign of the source number, the instruction increments SI. Then for *j* characters, the instruction either places a digit substitute in the destination field beginning at the position specified by DI, or it moves a digit 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, the instruction places both the digit substitute and the digit in the destination field, and increases SI by *j*. If T does not change from 0 to 1, the instruction increases DI by *j*. If T does change from 0 to 1, the instruction increases DI by *j*+1.

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

If T is 1, the instruction moves each digit processed from the source field to the destination field. If T is 0 and the digit is a zero or space, the instruction places *p0* in the destination field. If T is 0 and the digit is a non-zero, the instruction sets T to 1 and the characters placed in the destination field depend on S. If S is 0, the instruction places *p1* in the destination field followed by the digit. If S is 1, the instruction places *p2* in the destination field followed by the digit.

The instruction initiates a commercial fault if any of the digits processed is not valid for the specified data type.

Move Numerics

DMVN *j*

0	1	2	3	4	5	6	7
0	0	0	0	1	0	0	0

0	1	2	3	4	5	6	7
j							

Increments SI if the source data type is 3 and *j*>0. The instruction then moves *j* characters from the source field beginning at the position specified by SI to the destination field beginning at the position sp cifed by DI. Increases both SI and DI by *j*. Sets T to 1.

Initiates a commercial fault if any of the characters moved is not valid for the specified data type.

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

Move Digit With Overpunch

DMVO *p0,p1,p2,p3*

0	1	2	3	4	5	6	7
0	0	0	0	0	1	1	1

0	1	2	3	4	5	6	7
p0							

0	1	2	3	4	5	6	7
p1							

0	1	2	3	4	5	6	7
p2							

0	1	2	3	4	5	6	7
p3							

Increments SI if the source data type is 3 and SI points to the sign of the source number. The instruction then either places a digit substitute in the destination field (at the position specified by DI), or it moves a digit plus overpunch from the source field (at the position specified by SI) to the destination field (at the position specified by DI). Increases both SI and DI by 1.

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

If the digit is a zero or space and S is 0, then the instruction places *p0* in the destination field. If the digit is a zero or space and S is 1, then the instruction places *p1* in the destination field. If the digit is a non-zero and S is 0, the instruction adds *p2* to the digit and places the result in the destination field. If the digit is a non-zero and S is 1, the instruction adds *p3* to the digit and places the result in the destination field. If the digit is a non-zero, the instruction sets T to 1. The instructions assumes *p2* and *p3* are ASCII characters.

The instruction initiates a commercial fault if the character is not valid for the specified data type.

Move Numeric With Zero Suppression

DMVS *j,p0*

0	1	2	3	4	5	6	7
0	0	0	1	0	1	1	0

0	1	2	3	4	5	6	7
j							

0	1	2	3	4	5	6	7
p0							

Increments SI if the source data type is 3, *j*>0, and SI points to the sign of the source number. The instruction then moves *j* characters from the source field (beginning at the position specified by SI) to the destination field (beginning at the position specified by DI). Moves the digit from the source to the destination if T is 1. Replaces all zeros and spaces with *p0* as long as T is 0. Sets T to 1 when the first non-zero digit is encountered. Increases both SI and DI by *j*.

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

This op-code destroys the data type specifier.

Initiates a commercial fault if any of the characters moved is not a numeric (0-9 or space).

End Float

DNDF *p0,p1*

0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	1

0	1	2	3	4	5	6	7
p0							

0	1	2	3	4	5	6	7
p1							

If T is 1, the instruction places nothing in the destination field and leaves DI unchanged. If T is 0 and S is 0, the instruction places *p0* in the destination field at the position specified by DI. If T is 0 and S is 1, the instruction places *p1* in the destination field at the position specified by DI. It increases DI by 1, and sets T to one.

Set S To One

DSSO

0	1	2	3	4	5	6	7
0	0	0	0	0	1	0	1

Sets the Sign flag (S) to 1.

Set S To Zero

DSSZ

0	0	0	0	0	1	0	0
0	1	2	3	4	5	6	7

Sets the Sign flag (S) to 0.

Store In Stack

DSTK *k,p0*

0	0	0	0	0	0	1	0
0	1	2	3	4	5	6	7

k							
0	1	2	3	4	5	6	7

p0							
0	1	2	3	4	5	6	7

Stores the byte specified by *p0* in bits 8-15 of a word in the stack. Sets bits 0-7 of the word that receives *p0* to 0. If the 8-bit two's complement integer specified by *k* is negative, the instruction addresses the word receiving *p0* by (stack pointer+1+*k*). If *k* is positive, then the instruction stores *p0* at the address (frame pointer+1+*k*).

Set T To One

DSTO

0	0	0	0	1	0	0	1
0	1	2	3	4	5	6	7

Sets the significance Trigger (T) to 1.

Set T To Zero

DSTZ

0	0	0	0	0	1	1	0
0	1	2	3	4	5	6	7

Sets the significance Trigger (T) to 0.

Extended Decrement and Skip if Zero

EDSZ *[@]displacement[,index]*

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

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

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

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

Extended Increment And Skip If Zero

EISZ *[@]displacement[,index]*

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

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

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

Computes the effective address, *E* , increments the contents that of memory location by one, and writes the new value back into memory at the same address. If the updated value of the location is zero, the instruction increments the program counter by one and continues sequential operation at the updated value of the program counter.

Extended Jump

EJMP *[@]displacement[,index]*

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

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

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

Extended Jump To Subroutine

EJSR *[@]displacement[,index]*

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

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

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

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

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

Extended Load Accumulator

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

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

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

Moves a word out of memory and into an accumulator.

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

Extended Load Byte

ELDB *ac,displacement[,index]*

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

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

Copies a byte from memory into an accumulator.

Forms a byte pointer by first taking an index value, multiplying it by 2, and then adding the low-order 16 bits of the result to the displacement. Copies the byte addressed by this byte pointer into bits 8-15 of the specified accumulator, and sets bits 0-7 of that accumulator to 0. The instruction destroys the previous contents of the specified accumulator, but it does not alter either the index value or the displacement.

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

INDEX BITS	INDEX VALUE
00	0
01	Address of the displacement field (Word 2 of this instruction)
10	Contents of AC2
11	Contents of AC3

Load Effective Address

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

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

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

Places an effective address in an accumulator.

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

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

Extended Store Accumulator

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

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

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

Stores the contents of an accumulator into a memory location.

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

Extended Store Byte

ESTB *ac,displacement[,index]*

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

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

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

Forms a byte pointer by first taking an index value, multiplying it by 2, and then adding the low-order 16 bits of that result to the displacement. Copies bits 8-15 of the specified accumulator into memory at the byte address specified by the computed byte pointer. The instruction does not alter the specified accumulator.

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

INDEX BITS	INDEX VALUE
00	0
01	Address of the displacement field (Word 2 of this instruction)
10	Contents of AC2
11	Contents of AC3

Absolute Value

FAB *fpac*

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

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

Add Double (FPAC to FPAC)

FAD *facs,facd*

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

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

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

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

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

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

Add Double (Memory to FPAC)

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

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

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

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

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

Floating point addition consists of an exponent comparison and a mantissa addition. The exponents of the two numbers are compared, and the mantissa of the number with the smaller exponent is shifted right. This mantissa alignment is accomplished by taking the absolute value of the difference between the two exponents and shifting the mantissa right that number of hex digits. One guard digit is provided so that all but four bits shifted out of the right end of the mantissa are lost, and do not take part in the addition. If all significant digits are shifted out of the mantissa, the operation is equivalent to adding the number with the larger exponent to zero. This requires a shift of at least 15 hex digits for double precision, or 7 hex digits for single precision.

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

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

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

Add Single (Memory to FPAC)

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

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

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

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

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

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

If all significant digits are shifted out of the mantissa, the operation is equivalent to adding the number with the larger exponent to zero. This requires a shift of at least 15 hex digits for double precision, or 7 hex digits for single precision.

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

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

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

Add Single (FPAC to FPAC)

FAS *facs,facd*

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

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

Floating point addition consists of an exponent comparison and a mantissa addition. The exponents of the two numbers are compared, and the mantissa of the number with the smaller exponent is shifted right. This mantissa alignment is accomplished by taking the absolute value of the difference between the two exponents and shifting the mantissa right that number of hex digits. One guard digit is provided so that all but four bits shifted out of the right end of the mantissa are lost, and do not take part in the addition. If all significant digits are shifted out of the mantissa, the operation is equivalent to adding the number with the larger exponent to zero. This requires a shift of at least 15 hex digits for double precision, or 7 hex digits for single precision.

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

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

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

Clear Errors

FCLE

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

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

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

Compare Floating Point

FCMP *facs,facd*

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

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

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

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

NOTE: *Unnormalized operands give unspecified results.*

Divide Double (FPAC by FPAC)

FDD *facs,facd*

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

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

The source operand is checked for a zero mantissa. If the mantissa is zero, the DVZ bit is set in the floating point status register and the instruction is terminated. The number in FACD remains unchanged. If the mantissa is nonzero, the previous contents of FACD are lost. The two mantissas are compared and if the mantissa of the number in FACD is greater than or equal to the mantissa of the source operand, the mantissa of the number in FACD is shifted right one hex digit and the exponent of the number in FACD is increased by one. This process continues until the mantissa of the number in FACD is less than the mantissa of the source operand. Since one guard digit is provided, all but four bits shifted out are lost.

The mantissa in FACD is then divided by the mantissa of the source operand and the quotient is the mantissa of the intermediate result. The exponent of the source operand is subtracted from the exponent in FACD and 64 is added to this result. This addition of 64 maintains the *excess 64* notation. The result of the exponent manipulation becomes the exponent of the intermediate result. The sign of the intermediate result is determined from the sign of the two operands by the rules of algebra. The result is normalized and placed in FACD.

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

Divide Double (FPAC by Memory)

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

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

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

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

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

The source operand is checked for a zero mantissa. If the mantissa is zero, the DVZ bit is set in the floating point status register and the instruction is terminated. The number in FPAC remains unchanged. If the mantissa is nonzero, the previous contents of FPAC are lost. The two mantissas are compared and if the mantissa of the number in FPAC is greater than or equal to the mantissa of the source operand, the mantissa of the number in FPAC is shifted right one hex digit and the exponent of the number in FPAC is increased by one. This process continues until the mantissa of the number in FACD is less than the mantissa of the source operand. Since one guard digit is provided, all but four bits shifted out are lost.

The mantissa in FPAC is then divided by the mantissa of the source operand and the quotient is the mantissa of the intermediate result. The exponent of the source operand is subtracted from the exponent in FPAC and 64 is added to this result. This addition of 64 maintains the *excess 64* notation. The result of the exponent manipulation becomes the exponent of the intermediate result. The sign of the intermediate result is determined from the sign of the two operands by the rules of algebra. The result is normalized and placed in FPAC.

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

Divide Single (FPAC by Memory)

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

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

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

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

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

The source operand is checked for a zero mantissa. If the mantissa is zero, the DVZ bit is set in the floating point status register and the instruction is terminated. The number in FPAC remains unchanged. If the mantissa is nonzero, the previous contents of FPAC are lost. The two mantissas are compared and if the mantissa of the number in FPAC is greater than or equal to the mantissa of the source operand, the mantissa of the number in FPAC is shifted right one hex digit and the exponent of the number in FPAC is increased by one. This process continues until the mantissa of the number in FPAC is less than the mantissa of the source operand. Since one guard digit is provided, all but four bits shifted out are lost.

The mantissa in FPAC is then divided by the mantissa of the source operand and the quotient is the mantissa of the intermediate result. The exponent of the source operand is subtracted from the exponent in FPAC and 64 is added to this result. This addition of 64 maintains the *excess 64* notation. The result of the exponent manipulation becomes the exponent of the intermediate result. The sign of the intermediate result is determined from the sign of the two operands by the rules of algebra. The result is normalized and placed in FPAC.

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

Divide Single (FPAC by FPAC)

FDS *facs,facd*

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

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

The source operand is checked for a zero mantissa. If the mantissa is zero, the DVZ bit is set in the floating point status register and the instruction is terminated. The number in FACD remains unchanged. If the mantissa is nonzero, the previous contents of FACD are lost. The two mantissas are compared, and if the mantissa of the number in FACD is greater than or equal to the mantissa of the source operand, the mantissa of the number in FACD is shifted right one hex digit and the exponent of the number in FACD is increased by one. This process continues until the mantissa of the number in FACD is less than the mantissa of the source operand. Since one guard digit is provided, all but four bits shifted out are lost.

The mantissa in FACD is then divided by the mantissa of the source operand and the quotient is the mantissa of the intermediate result. The exponent of the source operand is subtracted from the exponent in FACD and 64 is added to this result. This addition of 64 maintains the *excess 64* notation. The result of the exponent manipulation becomes the exponent of the intermediate result. The sign of the intermediate result is determined from the sign of the two operands by the rules of algebra. The result is normalized and placed in FACD.

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

Load Exponent

FEXP *fpac*

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

Places bits 1-7 of AC0 in bits 1-7 of the specified FPAC. Ignores bits 0 and 8-15 of AC0. Leaves unchanged bits 0 and 8-63 of FPAC and the entire contents of AC0. Also sets bits 0-7 (the sign and exponent) to zero if bits 8-63 (the mantissa) of FPAC are zero. Leaves bits 1-7 of FPAC unchanged if FPAC contains true zero.

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

Fix To AC

FFAS *ac,fpac*

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

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

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

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

> NOTE: *If the lower 15 bits of the integer formed from the number in FPAC are all 0, the sign bit of the result will be zero, regardless of the sign of the original number.*

Fix To Memory

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

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

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

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

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

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

If the lower 31 bits of the integer formed from the number in FPAC are all 0, the sign bit of the result will be zero.

Halve

FHLV *fpac*

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

Divides the floating point number in FPAC by 2.

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

Integerize

FINT *fpac*

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

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

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

Float From AC

FLAS *ac,fpac*

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

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

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

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

Load Floating Point Double

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

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

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

Moves four words out of memory into a specified FPAC.

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

Load Floating Point Single

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

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

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

Moves two words out of memory into a specified FPAC.

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

Float From Memory

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

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

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

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

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

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

Load Floating Point Status

FLST *[@]displacement[,index]*

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

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

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

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

Multiply Double (FPAC by FPAC)

FMD *facs,facd*

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

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

The mantissas of the two numbers are multiplied together to give the mantissa of the intermediate result. One guard digit is provided for the intermediate result, which is used if normalization is required. The exponents of the two numbers are added together and 64 is subtracted. This subtraction of 64 maintains the *excess 64* notation. The result of the exponent manipulation becomes the exponent of the intermediate result. The sign of the intermediate result is determined from the 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 flag in the floating point status register is set to 1. The number is correct except that, for exponent overflow, the exponent is 128 too small, and for exponent underflow, the exponent is 128 too large.

Multiply Double (FPAC by Memory)

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

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

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

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

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

The mantissas of the two numbers are multiplied together to give the mantissa of the intermediate result. One guard digit is provided for the intermediate result, which is used if normalization is required. The exponents of the two numbers are added together and 64 is subtracted. This subtraction of 64 maintains the *excess 64* notation. The result of the exponent manipulation becomes the exponent of the intermediate result. The sign of the intermediate result is determined from the 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 flag in the floating point status register is set to 1. The number is correct except that, for exponent overflow, the exponent is 128 too small, and for exponent underflow, the exponent is 128 too large.

Multiply Single (FPAC by Memory)

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

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

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

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

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

The mantissas of the two numbers are multiplied together to give the mantissa of the intermediate result. One guard digit is provided for the intermediate result, which is used if normalization is required. The exponents of the two numbers are added together and 64 is subtracted. This subtraction of 64 maintains the *excess 64* notation. The result of the exponent manipulation becomes the exponent of the intermediate result. The sign of the intermediate result is determined from the 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 flag in the floating point status register is set to 1. The number is correct except that, for exponent overflow, the exponent is 128 too small, and for exponent underflow, the exponent is 128 too large.

Move Floating Point

FMOV *facs,facd*

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

Moves the contents of one FPAC to another FPAC.

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

Multiply Single (FPAC by FPAC)

FMS *facs,facd*

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

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

The mantissas of the two numbers are multiplied together to give the mantissa of the intermediate result. One guard digit is provided for the intermediate result, which is used if normalization is required. The exponents of the two numbers are added together and 64 is subtracted. This subtraction of 64 maintains the *excess 64* notation. The result of the exponent manipulation becomes the exponent of the intermediate result. The sign of the intermediate result is determined from the 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 flag in the floating point status register is set to 1. The number is correct except that, for exponent overflow, the exponent is 128 too small, and for exponent underflow, the exponent is 128 too large.

Negate

FNEG *fpac*

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

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

Normalize

FNOM *fpac*

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

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

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

No Skip

FNS

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

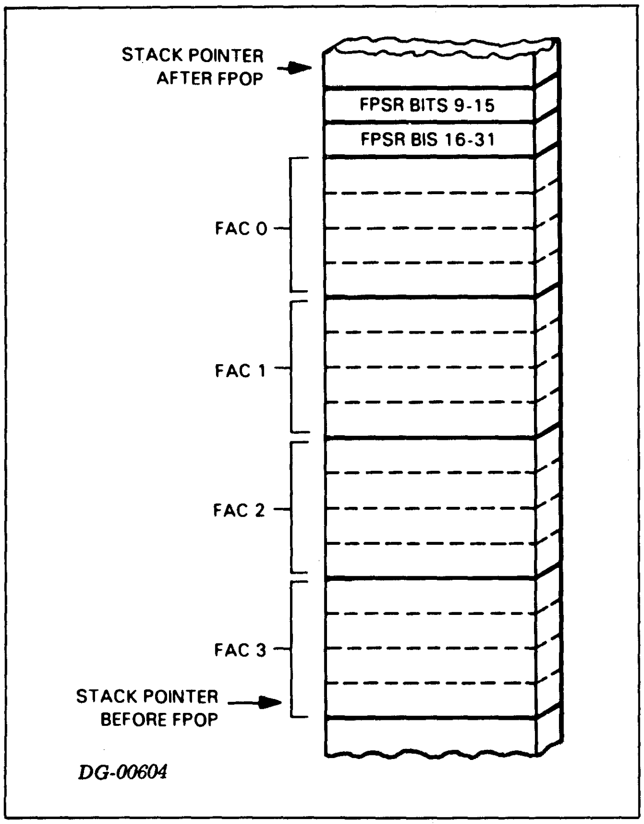
The next sequential word is executed.

Pop Floating Point State

FPOP

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

Pops an 18-word floating point return block off the user stack and alters the state of the floating point unit. The words popped and their destinations are as follows:



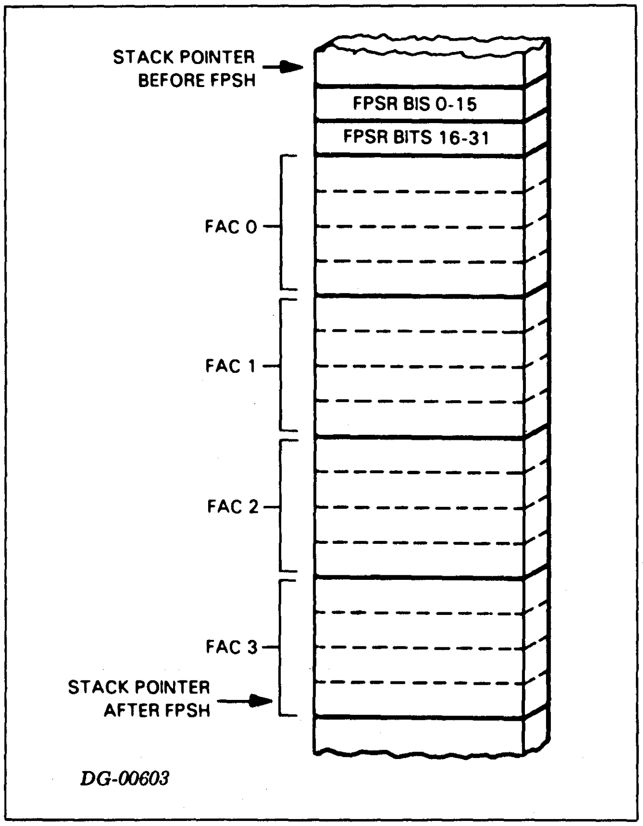
NOTE: *Because of the potentially long time required to perform some floating point instructions in relation to I/O interrupt requests, these instructions are interruptable. Because the FACD, stack pointer, and program counter are not updated until the completion of these instructions, any interrupt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

Push Floating Point State

FPSH

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

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



Read High Word

FRH *fpac*

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

Places the high-order 16 bits of FPAC in AC0, destroys the previous contents of AC0, and leaves unchanged the contents of FPAC and the Z and N flags in the floating point status register.

Skip Always

FSA

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

The next sequential word is skipped.

Scale

FSCAL *fpac*

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

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

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

Subtract Double (FPAC from FPAC)

FSD *facs,facd*

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

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

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

Skip On Zero

FSEQ

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

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

Skip On Greater Than Or Equal To Zero

FSGE

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

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

Skip On Greater Than Zero

FSGT

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

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

Skip On Less Than Or Equal To Zero

FSLE

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

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

Skip On Less Than Zero

FSLT

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

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

Subtract Double (Memory from FPAC)

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

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

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

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

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

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

Subtract Single (Memory from FPAC)

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

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

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

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

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

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

Skip On No Zero Divide

FSND

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

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

Skip On Non-Zero

FSNE

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

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

Skip On No Mantissa Overflow

FSNM

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

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

Skip On No Error

FSNER

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

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

Skip On No Overflow

FSNO

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

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

Skip On No Overflow and No Zero Divide

FSNOD

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

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

Skip On No Underflow

FSNU

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

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

Skip On No Underflow And No Zero Divide

FSNUD

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

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

Skip On No Underflow And No Overflow

FSNUO

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

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

Subtract Single (FPAC from FPAC)

FSS *facs,facd*

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

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

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

Store Floating Point Status

FSST *[@]displacement[,index]*

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

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

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

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

Store Floating Point Double

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

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

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

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

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

Store Floating Point Single

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

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

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

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

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

Trap Disable

FTD

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

Sets the trap enable bit of the FPSR to 0.

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

Trap Enable

FTE

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

Sets the trap enable bit of the FPSR to 1.

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

Halve

HLV *ac*

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

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

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

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

Hex Shift Left

HXL *n,ac*

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

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

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

Hex Shift Right

HXR *n,ac*

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

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

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

Increment

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

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

Increments the contents of an accumulator.

Initializes the carry bit to the specified value. Increments the unsigned, 16-bit number in ACS by one and places the result in the shifter. If the incrementation produces a carry of 1 out of the high order bit, the instruction complements the carry bit. Performs the specified shift operation, and loads the result of the shift into ACD if the no-load bit is 0. If the skip condition is true, the next sequential word is skipped.

> NOTE: *If the number in ACS is 177777_8 the instruction complements the carry bit.*

Inclusive OR

IOR *acs,acd*

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

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

Inclusive OR Immediate

IORI *i,ac*

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

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

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

Increment And Skip If Zero

ISZ *[@]displacement[,index]*

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

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

Increments the word addressed by *E* and writes the result back into memory at that location. If the updated value of the location is zero, the instruction places the address of the next sequential instruction in the program counter and operation continues from there.

Jump

JMP

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

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

Jump To Subroutine

JSR *[@]displacement[,index]*

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

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

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

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

Load Accumulator

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

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

Copies a word from memory to an accumulator.

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

Load Byte

LDB *acs,acd*

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

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

Places the 8-bit byte addressed by the byte pointer contained in ACS in bits 8-15 of ACD. Sets bits 0-7 of ACD to 0. The contents of ACS remain unchanged unless ACS and ACD are the same accumulator.

Load Integer

LDI *fpac*

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

Translates a decimal integer from memory to (normalized) floating point format and places the result in a floating point accumulator.

Under the control of accumulators AC1 and AC3, converts a decimal integer to floating point form, normalizes it, and places it in the specified FPAC. The instruction updates the Z and N bits in the FPSR to describe the new contents of the specified FPAC. Leaves the decimal number unchanged in memory, and destroys the previous contents of the specified FPAC.

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

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

Numbers of data type 7 are not normalized after loading. By convention, the first byte of a number stored according to data type 7 must contain the sign and exponent of the floating point number. The exponent must be in "excess 64" representation. The instruction copies each byte (following the lead byte) directly to mantissa of the specified FPAC. It then sets to zero each low-order byte in the FPAC that does not receive data from memory.

Upon successful completion, the instruction leaves accumulators AC0 and AC1 unchanged. AC2 contains the original contents of AC3; the contents of AC3 are undefined.

NOTE: *An attempt to load a minus 0 sets the specified FPAC to true zero.*

Load Integer Extended

LDIX

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

Distributes a decimal integer of data type 0, 1, 2, 3, 4, or 5 into the four FPACs.

Extends the integer with high-order zeros until it is 32 digits long. Divides the integer into four units of 8 digits each and converts each unit to a floating point number. Places the number obtained from the 8 high-order digits into FAC0, the number obtained from the next 8 digits into FAC1, the number obtained from the next 8 digits into FAC2, and the number obtained from the low-order 8 bits into FAC3. The instruction places the sign of the integer in each FPAC unless that FPAC has received 8 digits of zeros, in which case the instruction sets FPAC to true zero. The Z and N flags in the floating point status register are unpredictable.

AC1 must contain the data-type indicator describing 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 AC3 are undefined; the contents of AC1 remain unchanged; and AC2 contains the original contents of AC3.

Load Effective Address

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

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

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

If an auto-incrementing or auto-decrementing location is referenced in the course of the effective address calculation, the contents of the location are incremented or decremented. Note, however, that auto-incrementing and auto-decrementing is suppressed when demand paging is enabled.

The LEF instruction can only be used in a mapped system, while in the user mode. With the LEF mode bit set to 1, all I/O and LEF instructions will be interpreted as LEF instructions. With the LEF mode bit set to 0, all I/O and LEF instructions will be interpreted as I/O instructions.

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

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

Locate Lead Bit

LOB *acs,acd*

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

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

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

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

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

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

Logical Shift

LSH *acs,acd*

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

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

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

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

Load Sign

LSN

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

Under the control of accumulators AC1 and AC3, evaluates a decimal number in memory and returns in AC1 a code that classifies the number as zero or nonzero and identifies its sign. The meaning of the returned code is as follows:

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

AC1 must contain the data type indicator describing the number.

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

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.

Move

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

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

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

Initializes the carry bit to the specified value. Places the contents of ACS in the shifter. Performs the specified shift operation and loads the result of the shift into ACD if the no-load bit is 0. If the skip condition is true, the instruction skips the next sequential word.

Modify Stack Pointer

MSP *ac*

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

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

Adds the signed two's-complement number in AC to the stack pointer. If the result is less than the stack limit, the instruction places the result in the stack pointer.

If the result is greater than the stack limit, the instruction transfers control to the stack fault routine. The program counter in the fault return block is the address of the *Modify Stack Pointer* instruction. The stack pointer is left unchanged.

Unsigned Multiply

MUL

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

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

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

Signed Multiply

MULS

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

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

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

Negate

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

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

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

Initializes the carry bit to the specified value. Places the two's complement of the unsigned, 16-bit number in ACS in the shifter. If the negate operation produces a carry of 1 out of the high-order bit, the instruction complements the carry bit. Performs the specified shift operation and places the result in ACD if the no-load bit is 0. If the skip condition is true, the instruction skips the next sequential word.

> NOTE: *If ACS contains 0, the instruction complements the carry bit.*

Pop Multiple Accumulators

POP *acs,acd*

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

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

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

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

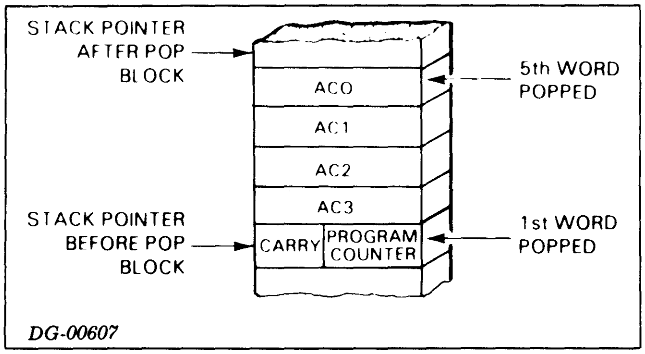
Pop Block

POPB

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

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

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



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

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

Pop PC And Jump

POPJ

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

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

Push Multiple Accumulators

PSH *acs,acd*

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

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

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

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

Push Jump

PSHJ *[@]displacement[,index]*

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

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

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

Push Return Address

PSHR

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

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

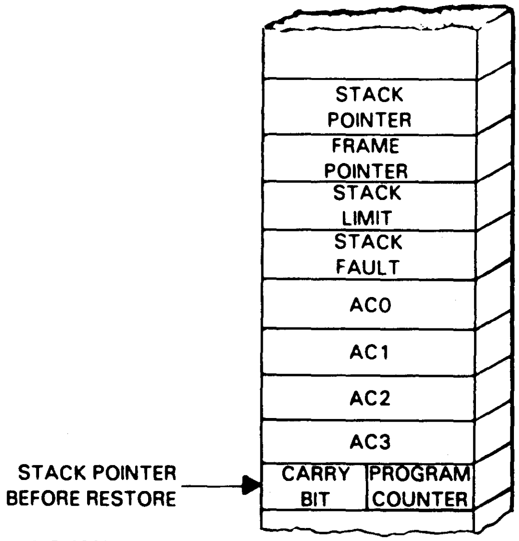
Restore

RSTR

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

Returns control from certain types of I/O interrupts.

Pops nine words off the stack and places them in predetermined locations. The words popped and their destinations are as follows:



DG-00606

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

> NOTE: *Use the* Restore *instruction to return control to the program only if the I/O interrupt handler uses the stack change facility of the* Vector on Interrupting Device Code *instruction.*
>
> *The* Restore *instruction does not check for stack underflow.*

Return

RTN

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

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

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 counter.

Save

SAVE *i*

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

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

Saves the information required by the *Return* instruction.

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 16-bit unsigned integer (called the *frame size*) contained in the immediate field is added to the stack pointer. 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

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

Before execution, the *Save* instruction checks for stack overflow. If executing the instruction would result in a stack overflow, *Save* transfers control to the stack fault routine. The program counter in the fault return block contains the address of the *Save* instruction.

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

Subtract Immediate

SBI *n,ac*

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

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

The contents of the immediate field *N*, plus 1 are subtracted from the unsigned 16-bit number contained in the specified AC and the result is placed in ACD. 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 code the exact value you wish to subtract.*

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

Skip If ACS Greater Than Or Equal to ACD

SGE *acs,acd*

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

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

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

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

Skip If ACS Greater Than ACD

SGT *acs,acd*

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

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

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

Skip On Non-Zero Bit

SNB *acs,acd*

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

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

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

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

Store Accumulator

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

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

Stores the contents of an accumulator into a memory location.

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

Store Byte

STB *acs,acd*

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

Moves the right byte of one accumulator to a byte in memory. The second accumulator contains the byte pointer.

Places bits 8-15 of ACD in the byte addressed by the byte pointer contained in ACS. The contents of ACS and ACD remain unchanged.

Store Integer

STI *fpac*

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

Under the control of accumulators AC1 and AC3, translates the contents of the specified FPAC to an integer of the specified type and stores it, right-justified, in memory, beginning at the specified location. The instruction leaves the floating point number unchanged in the FPAC, and destroys the previous contents of memory at the specified location(s).

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

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

Upon successful completion, the instruction leaves accumulators AC0 and AC1 unchanged. AC2 contains the original contents of AC3 and AC3 contains a byte pointer which is the address of the next byte after the destination field.

> NOTES: *If the number in the specified FPAC has any fractional part, the result of the instruction is undefined. Use the* Integerize *instruction to clear any fractional part.*
>
> *If the destination field cannot contain the entire number being stored, high-order digits are discarded until the number will fit into the destination. The remaining low-order digits are stored and Carry is set to 1.*
>
> *For data types 0, 1, 2, 3, 4, and 5, if the number being stored will not fill the destination field, the high-order bytes to the right of the sign are set to 0.*
>
> *For data type 6, if the number being stored will not fill the destination field, the sign bit is extended to the left to fill the field.*
>
> *For data type 7, if the number being stored will not fill the destination field, the low-order bytes are set to 0.*

Store Integer Extended

STIX

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

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

AC1 must contain the data-type indicator describing the form of the in memory.

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; the contents of 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.

> NOTES: *If the destination field is not large enough to contain the number being stored, the instruction disregards high-order digits until the number will fit in the destination. The instruction stores low-order digits remaining and sets the carry bit to 1.*
>
> *For data types 0, 1, 2, 3, 4, and 5, if the number being stored will not fill the destination field, the instruction sets the high-order bytes to 0.*
>
> *For data type 6, if the number being stored will not fill the destination field, the instruction extends the sign bit to the left to fill the field.*

Subtract

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

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

Performs unsigned integer subtraction and complements the carry bit if appropriate.

Initializes the carry bit to its specified value. The instruction subtracts the unsigned, 16-bit number in ACS from the unsigned, 16-bit number in ACD by taking the two's complement of the number in ACS and adding it to the number in ACD. The instruction places the result of the addition in the shifter. If the operation produces a carry of 1 out of the high-order bit, the instruction complements the carry bit. The instruction performs the specified shift operation and places the result of the shift in ACD if the no-load bit is 0. If the skip condition is true, the instruction skips the next sequential word.

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

System Call

SYC *acs,acd*

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

Pushes a return block and indirectly places the address of the *system call handler* in the program counter.

If a user map is enabled, the instruction disables it and pushes a return block onto the stack. The program counter in the return block points to the instruction immediately following the *System call* instruction. After pushing the return block, the instruction executed a *jump indirect* to location 2. 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, the instruction does not push a return block onto the stack. The contents of AC0 remain unchanged. If either of the accumulators specified is not AC0, then the instruction takes no special action. 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.

Skip On Zero Bit

SZB *acs,acd*

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

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

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

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

Skip On Zero Bit And Set To One

SZBO *acs,acd*

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

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

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

The instruction sets the addressed bit in memory to 1. If the bit was 0 before·being set to 1, the instruction skips the next sequential word. 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.*

Exchange Accumulators

XCH *acs,acd*

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

Exchanges the contents of two accumulators.

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

Execute

XCT *ac*

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

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

Because of 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 you a *wait for I/O interrupt* instruction.

> NOTE: *If the specified accumulator contains the first word of a two-word instruction, the word following the* XCT *instruction is used as the second word. Normal sequential operation then continues from the second word after the* XCT *instruction.*
>
> *The results of* XCT *are undefined if the specified accumulator contains an instruction that modifies that same accumulator. For example:*

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

Extended Operation

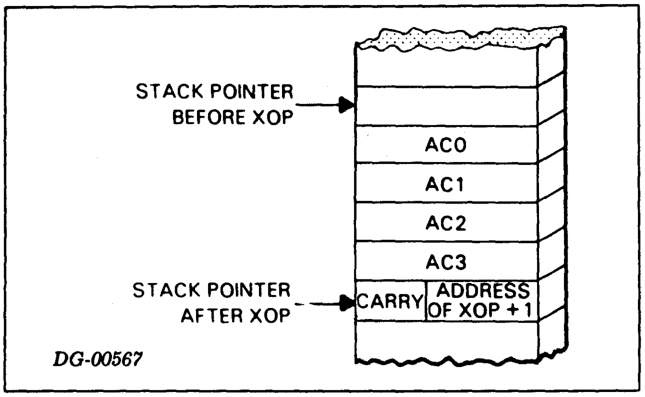
XOP *acs,acd,operation #*

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

Pushes a return block onto the stack. Places the address in the stack of ACS into AC2; places the address in the stack of ACD into AC3. Memory location 44_8 must contain the XOP origin address, the starting address of a 32_{10} word table of addresses. These addresses are the starting location of the various XOP operations.

Adds the operation number in the XOP instruction to the XOP origin address to produce the address of a word in the XOP table. The instruction fetches that word and treats it as the intermediate address in the effective address calculation. After the indirection chain, if any, has been followed, the instruction places the effective address in the program counter. The contents of AC0, AC1, and the XOP origin address remain unchanged.

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



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

Alternate Extended Operation

XOP1 *acs,acd,operation #*

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

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

Exclusive OR

XOR *acs,acd*

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

Forms the logical exclusive OR of the contents of ACS and the contents of ACD and places the result in ACD. Sets a bit position in the result to 1 if the corresponding bit positions in the two operands are unlike; otherwise, the instruction sets result bit to 0. The contents of ACS remain unchanged.

Exclusive OR Immediate

XORI *i,ac*

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

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

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

Chapter IV
ECLIPSE C/150 I/O INSTRUCTIONS

This chapter lists the ECLIPSE C/150 I/O instructions intended for a specific device such as the Map, the BMC, and special CPU instructions. We have arranged these instructions in alphabetical order according to mnemonics as recognized by the assembler.

For each instruction we include:

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

Some instructions can only be executed by the host processor, while others can also be executed by the I/O processor and/or the Data Control Unit. A label with each instruction indicates which processors can execute that instruction.

In general, these I/O instructions can be executed only with *Lef* mode and I/O protection disabled. See the Memory Allocation and Protection section in Chapter II for a discussion of *Lef* mode and I/O protection.

CODING AIDS

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

[] *[]* Square brackets indicate that the enclosed symbol (e.g., *[,skip]* is an optional operand or mnemonic. Code it only if you want to specify the option.

BOLD Code operands or mnemonics printed in boldface exactly as shown. For example, code the mnemonic for the *Move* instruction: **MOV**.

italic For each operand or mnemonic in italics, replace the item with a number or symbol that provides the assembler value you need for that item (e.g., the proper accumulator number, an address, etc.).

We use the following abbreviations throughout this chapter:

f or **F** Device Flag Command

AC or AC Accumulator

GENERAL I/O INSTRUCTIONS

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

Data In A

DIA *device*

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

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

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

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

Data in B

DIB *[f]* *ac,device*

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

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

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

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

Data In C

DIC *[f]* *ac,device*

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

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

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

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

Data Out A

DOA *[f] ac,device*

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

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

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

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

Data Out B

DOB *[f] ac,device*

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

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

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

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

Data Out C

DOC *[f] ac,device*

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

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

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

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

No I/O Transfer

NIO *[f] device*

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

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

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

I/O Skip

SKP *[t]* *device*

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

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

CENTRAL PROCESSOR

Device Code - 77_8 (Primary)

Priority Mask Bit - None

Device Flag Commands

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

f=S	Sets the interrupt enable flag to 1.
f=C	Sets the interrupt enable flag to 0.
f=P	If not an INTA instruction no effect. If the instruction is an INTA instruction, interprets the INTA instruction as the first word of a Vector instruction.
IORST	Sets the interrupt enable flag to 0.

Read Switches

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

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

Places the contents of the console switches into an accumulator.

Places the setting of the console data switches in the specified accumulator. After the transfer, sets the Interrupt On flag according to the function specified by F.

Interrupt Acknowledge

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

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

Returns device code of an interrupting device.

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

Mask Out

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

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

Sets the priority mask.

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

> NOTE: *A 1 in any bit disables interrupt requests at devices which use that bit as a mask.*

> NOTE: *Do not use this instruction when interrupts are enabled.*

Reset

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

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

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

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

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

Halt

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

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

Stops the processor.

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

> NOTE: *The assembler recognizes the mnemonic* HALT *as equivalent to the instruction* HALTA 0.

Interrupt Disable

INTDS
NIOC CPU

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

Sets Interrupt On flag to 0.

Interrupt Enable

INTEN
NIOS CPU

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

Sets Interrupt On flag to 1.

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

CPU Skip

SKP *[t]* CPU

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

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

See *Programmer's Reference-Peripherals (DGC No. 015-000021)* for a complete set of examples on using the interrupt system.

CPU Skip If Power Fail Flag Is One

SKPDN CPU

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

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

CPU Skip If Power Fail Flag Is Zero

SKPDZ CPU

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

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

Vector On Interrupting Device Code

VCT *[@]displacement[,index]*

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

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

Returns the device code of the interrupting device and uses that code as an index into a table. The value found in the table is then used as a pointer to the appropriate interrupt handler (Mode A) or as a pointer to another table which points to the interrupt handler and contains a new priority mask (Modes B through E). The instruction can also save the state of the machine by pushing various words onto the stack, creating a new vector stack, and setting up a priority structure.

The accompanying flow chart (*see opposite page*) is a complete diagram of the operation of the Vector instruction. Note that all modes use the *vector table* to find the next address used. Mode A uses the vector table entry as the address of the interrupt handler and passes control to it immediately. Modes B through E all use the vector table address as a pointer into a *device control table (DCT)*, where the address of the interrupt handler is found, along with a new priority mask.

Three control bits determine the mode of the *Vector* instruction which will be used. Their names and locations are:

Direct Bit - Bit 0 of the selected vector table entry;

Stack Change Bit - Bit 0 of the second word of the *Vector* instruction;

Push Bit - Bit 0 of the first word of the selected device control table.

The value of these bits determines the mode in which the *Vector*instruction operates, as shown following:

DIRECT	STACK	PUSH	MODE
0	don't care	don't care	A
1	0	0	B
1	0	1	C
1	1	0	D
1	1	1	E

The functions performed by the *Vector* instruction within each mode are summarized here:

MODE	FUNCTION
A	Uses device code returned by INTA as table entry to find address of interrupt handler.
B	Mode A plus: resets priority mask (saving old one) and reenables interrupts.
C	Mode B plus: pushes a normal 5-word return block (4 ACs, the program counter, and the carry bit) onto the stack.
D	Mode B plus: sets up a new vector stack for use by the interrupt handler and saves the old stack parameters.
E	Mode C plus Mode D.

In the following paragraphs, we will consider each mode and follow through the process step-by-step.

Common Process

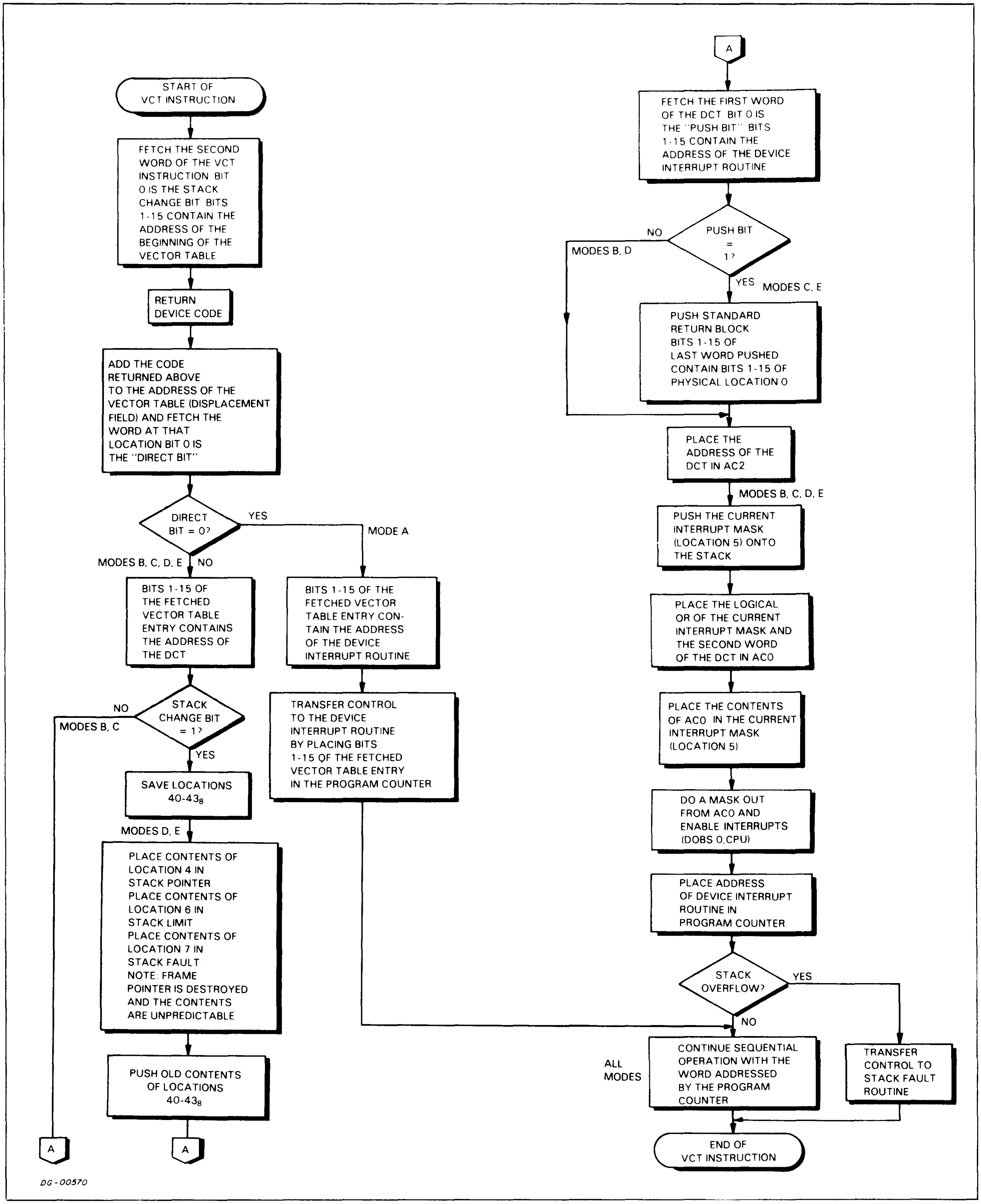
The initial steps taken by the *Vector* instruction are done regardless of the mode being used. The device code of the interrupting device is returned. This code is added to the address of the start of the vector table, which is found in the displacement field (bits 1-15 of the second instruction word), to get a new address within the vector table. The word at this new location is fetched and its bit 0 (the direct bit) is examined.

Mode A

If the direct bit is 0, mode A is used and the state of the other control bits does not matter. Bits 1-15 of the fetched vector table entry are used as the address of the interrupt handler for the interrupting device. Control is immediately transferred to the interrupt handler.

Mode B

Modes B through E perform different functions initially, but use a common second part. We discuss the common second part after discussing each Part I separately.


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
RETURN DEVICE CODE
ADD THE CODE RETURNED ABOVE TO THE ADDRESS OF THE VECTOR TABLE (DISPLACEMENT FIELD) AND FETCH THE WORD AT THAT LOCATION BIT 0 IS THE "DIRECT BIT"
DIRECT BIT = 0?
YES
MODE A
MODES B, C, D, E
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
MODES B, C
YES
SAVE LOCATIONS 40-43₈
MODES D, E
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 40-43₈
A
A
TRANSFER CONTROL TO THE DEVICE INTERRUPT ROUTINE BY PLACING BITS 1-15 OF THE FETCHED VECTOR TABLE ENTRY IN THE PROGRAM COUNTER
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
PUSH BIT = 1?
NO
MODES B, D
YES
MODES C, E
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
MODES B, C, D, E
PUSH THE CURRENT INTERRUPT MASK (LOCATION 5) ONTO THE STACK
PLACE THE LOGICAL OR OF THE CURRENT INTERRUPT MASK AND THE SECOND WORD OF THE DCT IN AC0
PLACE THE CONTENTS OF AC0 IN THE CURRENT INTERRUPT MASK (LOCATION 5)
DO A MASK OUT FROM AC0 AND ENABLE INTERRUPTS (DOBS 0,CPU)
PLACE ADDRESS OF DEVICE INTERRUPT ROUTINE IN PROGRAM COUNTER
STACK OVERFLOW?
YES
NO
ALL MODES
CONTINUE SEQUENTIAL OPERATION WITH THE WORD ADDRESSED BY THE PROGRAM COUNTER
TRANSFER CONTROL TO STACK FAULT ROUTINE
END OF VCT INSTRUCTION
DG-00570

Mode B - Part I

Mode B is used if the direct bit is 1 and the other two control bits are 0. The address in the vector table is now used as the location of the device control table *(DCT)* for the interrupting device. Bits 1-15 of the first word of the DCT contain the address of the desired interrupt handler (bit 0 is the push bit). The second word of the DCT is used to construct the new interrupt priority mask, and succeeding words (if any) contain information to be used by the device interrupt handler.

Mode C - Part I

If the direct bit and push bit are both 1, and the stack change bit is 0, mode C is used. The mode B functions are performed, and in addition, a standard 5-word return block is pushed onto the stack. This block consists of the contents of the four accumulators, the carry bit, and the contents of physical location 0 (the program counter return value).

Mode D - Part I

Mode D is used if the direct bit and the stack change bits are 1 and the push bit is 0. The mode B functions are performed, and in addition, a new stack is set up for the interrupt handler and the old contents of physical locations 40-43$_8$ (the user stack control words) are pushed onto the new stack.

Mode E - Part I

Mode E combines the functions of modes C and D. That is, the functions of mode B are performed, a new stack is set up, and a 5-word return block and the old stack control words are pushed onto the (new) stack.

Modes B through E - Part II

Modes B through E use the same procedure for the remainder of the *Vector* instruction. The current priority mask is pushed onto the stack. A *Mask Out* instruction is then performed, using the logical OR of the current mask and the second word of the DCT. The Interrupt On flag is set to 1 and control passes to the selected device interrupt handler. Note that the CPU permits one more instruction to execute (in this case, the first instruction of the interrupt handler) before the next I/O interrupt can occur.

ERCC ERROR CORRECTION

Device Code - 2$_8$ (Primary)

Priority Mask Bit - None

Device Flag Commands

f=S Sets the interrupt request flag and the Done flag to 0.

f=C No effect.

f=P No effect.

IORST Sets the interrupt request flag, the Done flag, and the ERCC control flags (bits 14 and 15) to 0; disables error checking and correction.

Read Memory Fault Address

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

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

Places the complement of bits 12-15 of the physical address of the memory location in error in bits 12-15 of the specified accumulator. Places the complement of bits 0-3 of that address in bits 0-3 of the accumulator. The previous contents of the specified AC are lost. The format of the specified AC is as follows:

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

BITS	NAME	CONTENTS or FUNCTION
0-3	PA 0-3	Complement of bits 0-3 of the physical address of the memory location in error.
4-11	---	Reserved for future use.
12-15	PA 12-15	Complement of bits 12-15 of the physical address of the memory location in error.

Read Memory Fault Code

DIB*[f]* *ac*,ERCC

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

Places a 5-bit error code in bits 0-4 of the specified accumulator. This code identifies the corrected bit. Sets bits 5-11 of the accumulator to 0 and places the complement of the four high-order bits of the physical address of the failing location in bits 12-15. The accumulator format is as follows:

FAULT												PA X0-X3			
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

BITS	NAME	CONTENTS or FUNCTION	
0-4	Code	A 5-bit code identifying which bit has an error	
		00000	No error
		00001	Check bit 4
		00010	Check bit 3
		00011	Data bit 0
		00100	Check bit 2
		00101	Data bit 1
		00110	Multiple bit error
		00111	Data bit 3
		01000	Check bit 1
		01001	Data bit 4
		01010	All 21 bits in memory are 1
		01011	Data bit 6
		01100	Data bit 7
		01101	Data bit 8
		01110	Data bit 9
		01111	Multiple bit error
		10000	Check bit 0
		10001	Data bit 11
		10010	Data bit 12
		10011	Data bit 13
		10100	Data bit 14
		10101	All 21 bits in memory are 0
		10110	Data bit 2
		10111	Multiple bit error
		11000	Data bit 10
		11001	Multiple bit error
		11010	Data bit 5
		11011	Multiple bit error
		11100	Data bit 15
		11101	Multiple bit error
		11110	Multiple bit error
		11111	Multiple bit error
5-11	----	Reserved for future use.	
12-15	PA X0-X3	Complement of bits X0-X3 of the physical address of the memory location in error.	

Enable ERCC

DOA*[f]* *ac*,ERCC

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

Enables the ERCC option according to the setting of bits 14-15 of the specified AC. Ignores bits 0-13 of the specified AC. The contents of the specified AC remain unchanged. The format of the specified AC is as follows:

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

BITS	NAME	CONTENTS or FUNCTION
0-13	----	Reserved for future use
14-15	ERCC	Control the ERCC feature as follows:
		00 Disable checking and correction; write valid check field.
		01 Disable checking and correction; for core memory, write check field with 1111; 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.

MEMORY ALLOCATION and PROTECTION

Device Code - 3_8 (Primary)

Priority Mask Bit - None

Device Flag Commands

f=S	No effect.
f=C	No effect.
f=P	Enables Map Single Cycle.
IORST	Disables Map.

Load Map

LMP

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

Under control of AC1 and AC2, loads successive words from memory into the MAP where they are used to define a user or data channel map.

AC1 must contain an unsigned integer, which is the number of words to be loaded into the MAP. Bits 1-15 of AC2 must contain the address of the first word to be loaded. If bit 0 of AC2 is 1, the instruction follows the indirection chain and places the resultant effective address in AC2. AC0 and AC3 are ignored and their contents remain unchanged.

For each word loaded, the instruction decrements the count in AC1 by one and increments the source address in AC2 by 1. Upon completion of the instruction, AC1 contains 0, and AC2 contains the address of the word following the last word loaded.

This instruction is interruptible in the same manner as the *Block add and move* instruction. If you issue this instruction while in mapped mode, with I/O protection enabled, the map will not be altered. AC1 and AC2 will be used and their contents modified as described above. No I/O trap will occur.

The words loaded into the MAP define the address translation functions for the various user and data channel maps. The contents of the MAP field (bits 6-8) of the MAP status register determine which map is affected by the *Load map* instruction. You can alter this field using either the *Load map status* or the *Initiate page check* instruction.

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

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

BITS	NAME	CONTENTS or FUNCTION
0	WP	Unused for data channel maps; write protect for user maps.
1-5	LOGICAL	Logical page number.
6-15	PHYSICAL	Physical page number.

NOTE: *Declare a logical page invalid by setting the Write Protect bit to 1 and all of bits 6-15 to 1.*

Read Map Status

DIA*[f]* *ac*,MAP

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

Reads the status of the current map.

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

		I/O	WP	IND	SC	MAP			LEF	I/O	WP	IND	A/B	DCH	U/M
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

BITS	NAME	CONTENTS or FUNCTION
0-1	---	Reserved for future use.
2	I/O	If 1, the last protection fault was an I/O protection fault.
3	WP	If 1, the last protection fault was a write protection fault.
4	IND	If 1, the last protection fault was an indirect protection fault.
5	Single Cycle	If 1, the last map reference was a *Map Single Cycle* instruction.
6-8	Map	Indicates which map will be loaded by 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 was enabled by the last *Load Map Status* instruction.
10	I/O	If 1, I/O protection was enabled by the last *Load Map Status* instruction.
11	WP	If 1, write protection was enabled by the last *Load Map Status* instruction
12	IND	If 1, indirect protection was enabled by the last *Load Map Status* instruction.
13	A/B	If 0, the last *Load Map Status* instruction enabled map A. If 1, the last *Load Map Status* instruction enabled user map B.
14	DCH Enable	If 1, the mapping of the data channel addresses is enabled.
15	User Mode	If 1, the last I/O interrupt occurred while in user mode.

Page Check

DIC *ac*,MAP

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

Provides the identity and some characteristics of the physical page corresponding to the logical page identified by the immediately preceding *Initiate Page Check* instruction.

Places the number of the physical page which corresponds to the logical page specified by the preceding *Initiate Page Check* or *Load Map Status* instruction in bits 6-15 of the specified AC. Places additional information about this page in bits 0-3 and destroys the previous contents of the AC. The format of the information placed in the specified AC is as follows:

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

BITS	NAME	CONTENTS or FUNCTION
0	WP	The write protect bit for the logical page which corresponds to the physical page specified by bits 6-15.
1-3	Map	The map which was used to perform the translation between logical page number and physical page number is 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.
6-15	Physical	The number of the physical page which corresponds to the logical page given in the preceding INITIATE PAGE CHECK instruction. If all these bits are 1, and WP (bit 0) is 1, then the logical page is validity protected.

Load Map Status

DOA *ac*,MAP

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

Defines the parameters of a new map.

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

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
						MAP			LEF	I/O	WP	IND	A/B	DCH	U/M

BITS	NAME	CONTENTS or FUNCTION
0-5	---	Reserved for future use.
6-8	MAP SEL	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, mapping of CPU addresses will commence with the first memory reference *after* the next *indirect* reference or return type instruction (POPB, POPJ, RTN, RSTR)

NOTE: *If the* Load Map Status *instruction sets the User Enable bit to 1, this inhibits the interrupt system and the MAP waits for either an indirect reference or return type instruction. Either event releases the interrupt system and allows the MAP to begin translating addresses (using the user map specified by bit 13 of the MAP status register). Address translation resumes (1) after the first level of the next indirect reference; or (2) after the first* Pop Block, Pop Jump, Return, *instruction that does not cause a stack fault.*

Map Page 31

DOB *ac*,MAP

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

Specifies that mapping take place for a single page of an unmapped address space. Mapping is always done for locations 76000$_8$ through 77777$_8$ (logical page 31). This is the only page which can be mapped when in unmapped address space. You can use this instruction to access a page of a user's memory space when in unmapped mode.

Bits 6-15 of the specified AC are transferred to the MAP. These bits specify a physical page number to which logical page 31 will be mapped when in the unmapped mode.

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

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

BITS	NAME	CONTENTS or FUNCTION
0-5	---	Reserved for future use.
6-15	Physical	The number of the physical page to which logical page 31 should be mapped when in unmapped mode.

Initiate Page Check

DOC *ac*,MAP

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

Identifies a logical page. The *Page Check* instruction will find the corresponding physical page.

Transfers the contents of the specified AC to the MAP for later use by the *Page Check* or *Load Map* instruction. Leaves the contents of the specified AC unchanged. The format of the specified AC is as follows:

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

BITS	NAME	CONTENTS or FUNCTION
0	---	Reserved for future use.
1-5	Logical Page	Number of the logical block for which the check is requested.
6-8	Map	Specify which map should be used for the check as follows: 000 User A 001 Reserved 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.

Map Single Cycle

Disable User Mode

NIOP *ac*,MAP

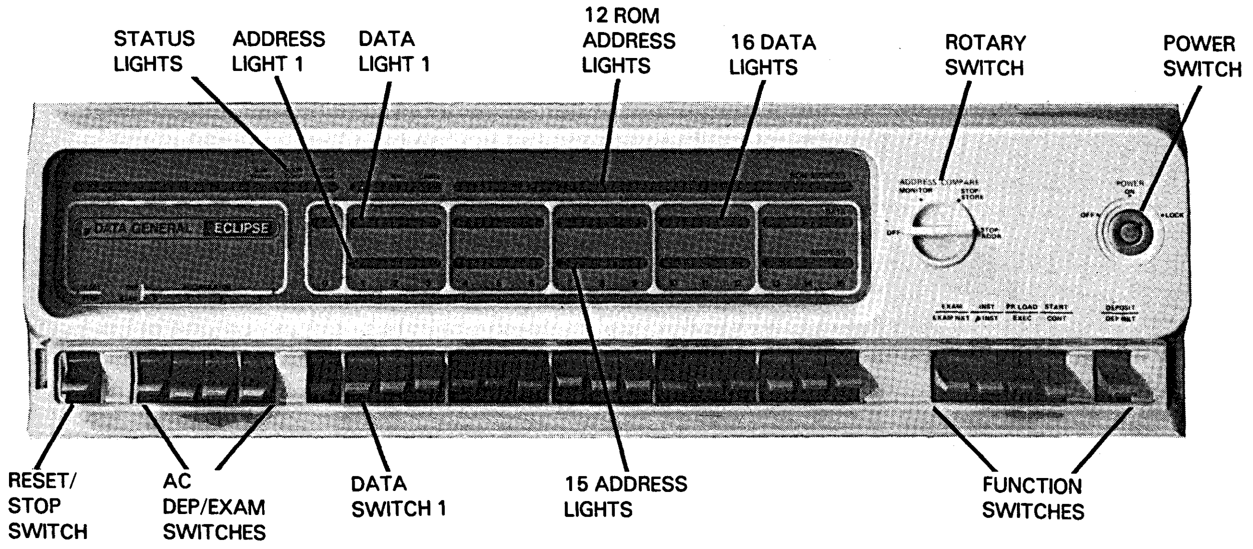
0	1	1	0	0	0	0	0	1	1	0	0	0	0	1	1
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Issued from unmapped mode, the instruction maps one memory reference using the last user map; issued from User mode with LEF mode and I/O protection disabled, the instruction simply turns off the map, returning it to unmapped mode. It is used by the supervisor to access a user's memory space when only one or two references are required. It is also used by a privileged user to turn off memory mapping.

From unmapped mode - Enables the user map for one memory reference. Maps the first memory reference of the next LDA, ELDA, STA or ESTA instruction. After the memory cycle is mapped, the instruction again disables the user map.

> NOTE: *The interrupt system is disabled from the beginning of the* Map single cycle *instruction until after the next* LDA, ELDA, STA *or* ESTA *instruction.*

From User mode - If LEF Mode and I/O protection is disabled, this instruction turns off the MAP. All subsequent memory references are unmapped until the map is reactivated with a *Load map status* instruction.



FUNCTION SWITCHES

NAME	POSITION	FUNCTION
Exam/ Exam N	Up	Loads PC with value of data switches, and displays contents of that address. Also fills MD register. To use while processor is running, Address Compare switch must be set to Mon.
	Down	Increments PC, and displays contents of that address.
Dep/ Dep N	Up	Deposits value in data switches at PC address
	Down	Increments PC and deposits value in data switches in that address.
St/ Cont	Up	Loads with value in data switches and starts normal execution. Also fills MD register.
	Down	Initiates normal operation from the current state of the machine.
Rest/ Stp	Up	Sets to 0 all Busy and Done flags and status lights, except Carry. ROM address lights display 0002_8.
	Down	Halts the CPU.
AC Dep/ Exam	Up	Loads the associated accumulator with the value in the data switches.
	Down	Displays the contents of the associated accumulator.
Inst/ U Inst	Up	Executes one machine instruction then halts the CPU.
	Down	Freezes the CPU after executing one microinstruction. Address lights show output of ALU.
P Load Exec	Up	Loads bootstrap loader program. Data switches 10-15 contain device code, and switch 0 is 1 if device is on DCH
	Down	Executes instruction contained in data switches.

ROTARY SWITCH

NAME	SETTING	EFFECTS
Address Compare	Off	Has no effect on processor operation.
	Monitor	When address in data switches is accessed, displays contents in data lights.
	Stop Store	When contents of address in data switches is changed, processor freezes.
	Stop Addr	When address in data switches is addressed, processor freezes.

STATUS LIGHTS

NAME	MEANING WHEN LIT
ION	I/O Interrupt flag is enabled.
Carry	Carry bit is 1.
ROM Error	Parity error in ROM is detected.*
User	MAP is in user mode. (MAP A)
Addr Compare	The conditions for a Monitor, Stop on Store or a Stop on Address have been met.

* If a ROM parity error occurs, the CPU freezes.

ADDRESS AND DATA LIGHTS

NAME	MEANING
ROM Address	Displays the address of the microinstruction last executed.
Data Lights	Displays contents of MEM Bus, except in Monitor mode.
Address	Displays contents of the address bus or the PC when halted.

CHAPTER V
CONSOLE FUNCTIONS

The console is a molded plastic panel with lights and switches that display and change the state of the machine. The position on the console and the general function of each of these lights and switches is shown in the removable diagram that precedes this page. There are five types of switches:

- A data switch (also called a toggle switch) -- has two positions. Up corresponds to 1, and down means 0.
- A function switch -- has three positions: up, down and neutral. When pushed up or down, it initiates a function; when released, it returns to the neutral position.
- A rotary switch -- may have any number of positions; once set to a position it remains there until manually altered.
- A lock -- has three positions and cannot be changed without the key.

Throughout the rest of the section we refer to each of these types of switches by the name given above or by the name of the function that switch performs. However, each *data* switch has its own name (0-15), which can be seen immediately above it. We use those names to specify some subset of all data switches. The same name also refers to the data light and address light that is immediately above each switch. The console diagram shows the relationship for data light, address light, and data switch 1.

While it is powered up, the CPU is always in one of three states: normal execution, frozen, or halted. When it is in normal execution, the microcode continually executes machine instructions from a program.

When the CPU is frozen, it does not execute microcode and it will not change state without external intervention. While in this state most of the console switches are disabled.

When the CPU is halted, it executes a small microinstruction loop (the ROM address lights display 0002 $_8$, and all of the console switches function normally. The CPU is in the halt state when it is powered up.

MAIN POWER PANEL

NAME	FUNCTION	OPERATION
CPU Power	Power	A key switch with positions labeled OFF, ON, and LOCK. With the switch in the OFF position, all power is removed from the CPU and the machine will not run. Turning the switch ON applies power, performs a RESET function, and enables all other console switches. Turning the switch to LOCK enables the key to be removed, and also disables all console switches except the MONITOR function of the ADDRESS COMPARE feature.

ROTARY SWITCH

NAME	POSITION	EFFECTS OF SETTING
ADDR COMPARE	OFF	Has no effect on processor operation.
	MONITOR	Displays contents of selected location in data lights, if and when that location is accessed. The setting of the data switches specifies the address of the monitored location. Updates the contents of data lights each time that location is accessed. NOTE: Data lights remain unchanged until monitor conditions are met. If that address is never accessed, the data lights will never display its contents.
	STOP STORE	Freezes processor when the contents of the selected location are altered. The setting of the data switches specifies the address of the selected location. Completes the store prior to the freeze.
	STOP ADDR	Freezes processor when the selected location is accessed. The setting of the data switches specifies the address of the selected location. The location is neither read nor written.

FUNCTION SWITCHES

NAME	POSITION	FUNCTION	MACHINE STATE*	MEANING
EXAM/ EX NEXT	UP	EXAMINE	HALTED	Loads PC with the logical address contained in data switches 1-15. Displays contents of that location in data lights, and displays address of that location in address lights.
			RUNNING	Displays contents of memory at location addressed by data switches. The Address Compare switch must be set to Monitor or Stop on Store for the display to remain long enough to be read. A running examine will not change the PC.
	DOWN	EXAMINE NEXT	HALTED	Increments PC, and uses that number as an address. Displays the contents of that address in data lights. Displays address of that location in address lights.
DEP/ DP NEXT	UP	DEPOSIT	HALTED	Stores the value contained in the 16 rightmost data switches (0-15) into the location addressed by PC. Displays new value of that location in data lights, and displays address of that location in the address lights.
	DOWN	DEPOSIT NEXT	HALTED	Increments PC and uses that number as an address to store value contained in the 16 rightmost data switches (0-15). Displays new value of that location in data lights, and displays address of that location in address lights.
START/ CONT	UP	START	HALTED	Loads the contents of the 15 rightmost data switches into PC, and executes the instruction at that address. Normal execution continues from there. Displays the last contents of the memory bus in data lights, and displays the contents of the selected address bus in address lights.
	DOWN	CONTINUE	HALTED, FROZEN	Initiates normal operation of the CPU from the current state of the machine.
RESET/ STOP	UP	RESET	RUNNING, FROZEN, HALTED	Stops the CPU immediately, initiates the equivalent of an I/O Reset instruction, setting the Busy and Done flags of all peripherals to 0. Sets all status lights on the console, except Carry, to 0. The ROM address lights will display 0002_8 (the halt location). The contents of the data and address lights are undefined. NOTE: The PC is unchanged; however, the instruction addressed by the current PC value may not have completed execution. This is the only function switch that will halt the CPU in the middle of an instruction.
	DOWN	STOP	RUNNING	Halts the CPU after the current instruction has been executed. Displays the address of the next instruction to be executed in address lights. Displays the last contents of the memory bus in data lights. The ROM address lights will show 0002_8 (the halt location). NOTE: Data channel requests will be honored after the halt, but interrupt requests will not be honored after the Stop function has been initiated.
DEP AC/ EXAM AC**	UP	DEPOSIT	HALTED	Loads the associated accumulator with the value contained in the 16 rightmost data switches (0-15). Displays the new contents of the AC in data lights.
	DOWN	EXAMINE	HALTED	Displays the contents of the associated accumulator in data lights.

FUNCTION SWITCHES

NAME	POSITION	FUNCTION	MACHINE STATE*	MEANING
INST/ µINST	UP	STEP INSTRUCTION	HALTED, FROZEN, RUNNING	Executes one machine instruction; then halts the processor. Displays the contents of the memory bus in data lights, and displays the address of the next instruction to be executed in address lights.
	DOWN	STEP MICRO-INSTRUCTION	HALTED, RUNNING	Executes one microinstruction; then freezes the CPU. Displays the contents of the MEM bus in the data lights; displays the output of the ALU bus in the address lights. Displays the address of the last microinstruction executed in the ROM address lights.
PROG LOAD/ EXEC	UP	BOOTSTRAP LOAD	HALTED	Executes a microdiagnostic program; then loads bootstrap loader program into memory locations 0-37$_8$, and executes it. If data switch 4 is 1 microdiagnostic will not be executed. Data switches 10-15 must contain the device code of the I/O device that contains the program to be loaded. If that device is on the data channel or the burst multiplexor channel, data switch 0 must be set to 1.
	DOWN	EXECUTE	HALTED	Executes instruction contained in 16 rightmost data switches (0-15), and halts the CPU. (Execute may be used with step microinstruction) NOTE: PC will be updated but the instruction at the old PC address will not be executed.

* If a function definition has no entry for a particular machine state, that function has no effect when in that state.

** There are 4 AC Dep/Exam switches on the console. Each performs the same functions on a different accumulator.

DG-04996

This page intentionally left blank.

APPENDIX A
STANDARD I/O DEVICE CODES

OCTAL DEVICE CODES	MNEMONIC	PRIORITY MASK BIT	DEVICE NAME	OCTAL DEVICE CODES	MNEMONIC	PRIORITY MASK BIT	DEVICE NAME
00	----	--	Unused	41[3]	DPO	8	IPB full duplex output
01	----	--	Unused	40	SCR	8	Synch. communication receiver
02	ERCC	--	Error checking and correction	41	SCT	8	Synch. communication transmitter
03	MAP	--	Memory allocation and protection unit	42	DIO	7	Digital I/O
04	DPM		Demand Paging Map	43	DIOT	6	Digital I/O timer
	IOPI	2,5	IOP Map, IDP Timer		PIT	6	Programmable Interval Timer
05				44	MXM	12	Modem control for MX1/MX2
06	MCAT	• 12	Multiprocessor adapter transmitter	45			
07	MCAR	12	Multiprocessor adapter receiver	46	MCAT1	12	Second multiprocessor transmitter
10	TTI	14	TTY input	47	MCAR1	12	Second multiprocessor receiver
11	TTO	15	TTY output	50	TTI1	14	Second TTY input
12	PTR	11	Paper tape reader	51	TTO1	15	Second TTY output
13	PTP	13	Paper tape punch	52	PTR1	11	Second paper tape reader
14	RTC	13	Real-time clock	53	PTP1	13	Second paper tape punch
15	PLT	12	Incremental plotter	54	RTC1	13	Second real-time clock
16	CDR	10	Card reader	55	PLT1	12	Second incremental plotter
17	LPT	12	Line printer	56	CDR1	10	Second card reader
20	DSK	9	Fixed head disc	57	LPT1	12	Second line printer
21	ADCV	8	A/D converter	60	DSK1	9	Second fixed head disc
22	MTA	10	Magnetic tape	61	ADCV1	8	Second A/D converter
23	DACV	--	D/A converter	62	MTA1	10	Second magnetic tape
24	DCM	0	Data communications multiplexor	63	DACV1	--	Second D/A converter
25				64			
26	DKB	9	Fixed head DG/Disc	65	IOP I	5[5]	Host To IOP Interface
27	DPF	7	DG/Disc storage subsystem	66	DKB1	9	Second Fixed Head DG/Disc
30	QTY	14	Asynch. hardware multiplexor	67	DPF1	7	Second DG/Disc storage subsystem
30	SLA	14	Synchronous line adapter	70	QTY1	14	Second asynch. hardware mux
31[1]	IBM1	13	IBM 360/370 interface	70	SLA1	14	Second synchronous line adapter
32	IBM2	13	IBM 360/370 interface	71[1]		13	Second IBM 360/370 interface
33	DKP	7	Moving head disc	72		13	Second IBM 360/370 interface
34[1]	CAS[1]	10	Cassette tape	73	DKP1	7	Second moving head disc
	DCU[4]	4	Data Control Unit				
34	MX1	11	Multiline asynchronous controller	74	CAS1	10	Second cassette tape
35	MX2	11	Multiline asynchronous controller	74[1]		11	Second multiline asynch. controller
36	IPB	6	Interprocessor bus--half duplex	75		11	Second multiline asynch. controller
37	IVT	6	IPB watchdog timer	76	DPU	4	DCU To Host Interface
40[2]	DPI	8	IPB full duplex input	77	CPU	--	CPU and console functions

[1] Code returned by INTA and used by VCT

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

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

[4] Can be set to any unused device code between 1 and 76

[5] Micro interrupts are not maskable

APPENDIX 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.

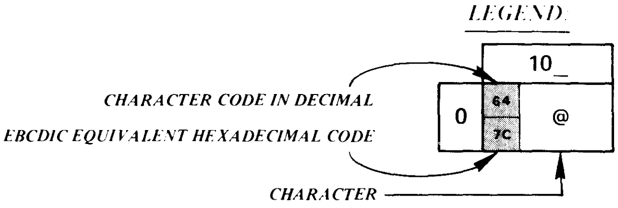
OCTAL CONVERSION TABLE

	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

HEXADECIMAL CONVERSION TABLE

	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



↥ *means* CONTROL

OCTAL	00_			01_			02_			03_			04_			05_			06_			07_		
0	0	00	NUL	8	16	BS *(BACK-SPACE)*	16	10	DLE ↥P	24	18	CAN ↥X	32	40	SPACE	40	4D	(	48	F0	Ø	56	F8	8
1	1	01	SOH ↥A	9	05	HT *(TAB)*	17	11	DC1 ↥Q	25	19	EM ↥Y	33	5A	!	41	5D	)	49	F1	1	57	F9	9
2	2	02	STX ↥B	10	15	NL *(NEW LINE)*	18	12	DC2 ↥R	26	3F	SUB ↥Z	34	7F	” *(QUOTE)*	42	5C	*	50	F2	2	58	7A	:
3	3	03	ETX ↥C	11	0B	VT *(VERT. TAB)*	19	13	DC3 ↥S	27	27	ESC *(ESCAPE)*	35	7B	#	43	4E	+	51	F3	3	59	5E	;
4	4	37	EOT ↥D	12	06	FF *(FORM FEED)*	20	3C	DC4 ↥T	28	1C	FS ↥\	36	5B	$	44	6B	, *(COMMA)*	52	F4	4	60	4C	<
5	5	2D	ENQ ↥E	13	0D	RT *(RETURN)*	21	3D	NAK ↥U	29	1D	CS ↥]	37	6C	%	45	60	—	53	F5	5	61	7E	=
6	6	2E	ACK ↥F	14	0E	SO ↥N	22	32	SYN ↥V	30	1E	RS ↥↑	38	50	&	46	4B	. *(PERIOD)*	54	F6	6	62	6E	>
7	7	2F	BEL ↥G	15	0F	SI ↥O	23	26	ETB ↥W	31	1F	US ↥←	39	7D	' *(APOS)*	47	61	/	55	F7	7	63	6F	?

OCTAL	10_			11_			12_			13_			14_			15_			16_			17_		
0	64	7C	@	72	C8	H	80	D7	P	88	E7	X	96	79	` *(GRAVE)*	104	88	h	112	97	p	120	A7	x
1	65	C1	A	73	C9	I	81	D8	Q	89	E8	Y	97	81	a	105	89	i	113	98	q	121	A8	y
2	66	C2	B	74	D1	J	82	D9	R	90	E9	Z	98	82	b	106	91	j	114	99	r	122	A9	z
3	67	C3	C	75	D2	K	83	E2	S	91	8D	[	99	83	c	107	92	k	115	A2	s	123	C0	{
4	68	C4	D	76	D3	L	84	E3	T	92	E0	\	100	84	d	108	93	l	116	A3	t	124	4F	|
5	69	65	E	77	D4	M	85	E4	U	93	9D	]	101	85	e	109	94	m	117	A4	u	125	D0	}
6	70	C6	F	78	D5	N	86	E5	V	94	5F	↑ or ∧	102	86	f	110	95	n	118	A5	v	126	A1	~ *(TILDE)*
7	71	C7	G	79	D6	O	87	E6	W	95	6D	← or_	103	87	g	111	96	o	119	A6	w	127	07	DEL *(RUBOUT)*

CHARACTER CODE IN OCTAL AT TOP AND LEFT OF CHARTS.

APPENDIX D
BINARY, OCTAL AND DECIMAL NUMBERING SYSTEMS

The most familiar numbering system in our society is the decimal system. For ordinary mental or pencil-and-paper work it is clearly the easiest to use. Computers, however, use the binary system, which becomes very confusing to humans when more than a few digits are involved. Fortunately, binary can be easily translated into octal or hexadecimal representation, both of which are relatively easy for humans to use.

In this section, we provide some basic background on the binary, octal and hexadecimal numbering systems. Most readers will already be familiar with these, but some may not and others may find the review helpful.

The binary numbering system is used in computers because the two binary values can be easily represented electronically. In the binary system, the only two permissible digits are 0 or 1, and each position in a binary number represents some power of 2. For example, consider the binary number:

1011010_2

which is equivalent to the sum (in decimal):

$(1\text{x}2^6) + (0\text{x}2^5) + (1\text{x}2^4) + (1\text{x}2^3) +$
$(0\text{x}2^2) + (1\text{x}2^1) + (0\text{x}2^0)$

or

$64 + 0 + 16 + 8 + 0 + 2 + 0 = 90_{10}.$

If we divide this number into groups of 3 starting at the right, thus:

1 011 010,

we see that each group of 3 has a range of:

000 = 0

to

$111 = 7 = (2^2 + 2^1 + 2^0) = (4 + 2 + 1).$

Zero to 7 is the range of digits allowable in the octal numbering sytem, so we can convert from binary to octal simply by grouping and evaluating each group of 3 binary digits by itself. In octal, the number above becomes:

1 011 010

or

$1 \quad 3 \quad 2 = 132_8$

We can also convert this number to hexadecimal (or base 16). Zero through nine *decimal* are unchanged in the hexadecimal system, but $10\text{-}15_{10}$are represented by the letters A through F.

If we divide the original binary number into groups of 4 instead of 3, starting from the right, we get:

101 1010

The range for one group is now:

0000 = 0

to

$1111 = 2^3 + 2^2 + 2^1 + 2^0$
$= (8 + 4 + 2 + 1) = 15_{10} = \text{F}_{16}$

The number in the example above is then:

101 1010

or

$5 \quad \text{A} = 5\text{A}_{16}$

APPENDIX E
COMPATIBILITY WITH NOVA LINE COMPUTERS

The ECLIPSE M/600 computers are compatible with Data General's NOVA line of computers. Any program presently running on any NOVA line computer will run on an ECLIPSE series computer without change provided that it does not violate any of the following constraints:

- The program may not be dependent on instruction execution times or Input/Output (I/O) transfer times. Times for the ECLIPSE series computers may be faster than a NOVA line computer depending upon the application.

- The program may not use any fixed-point arithmetic instructions that have both the *no-load* and *no-skip* options specified. The ECLIPSE series computers use these codes to implement instructions in the standard instruction set.

- The program may not require the hardware multiply/divide option available on any NOVA line computer.

- The program may not utilize the data channel increment or add-to-memory features.

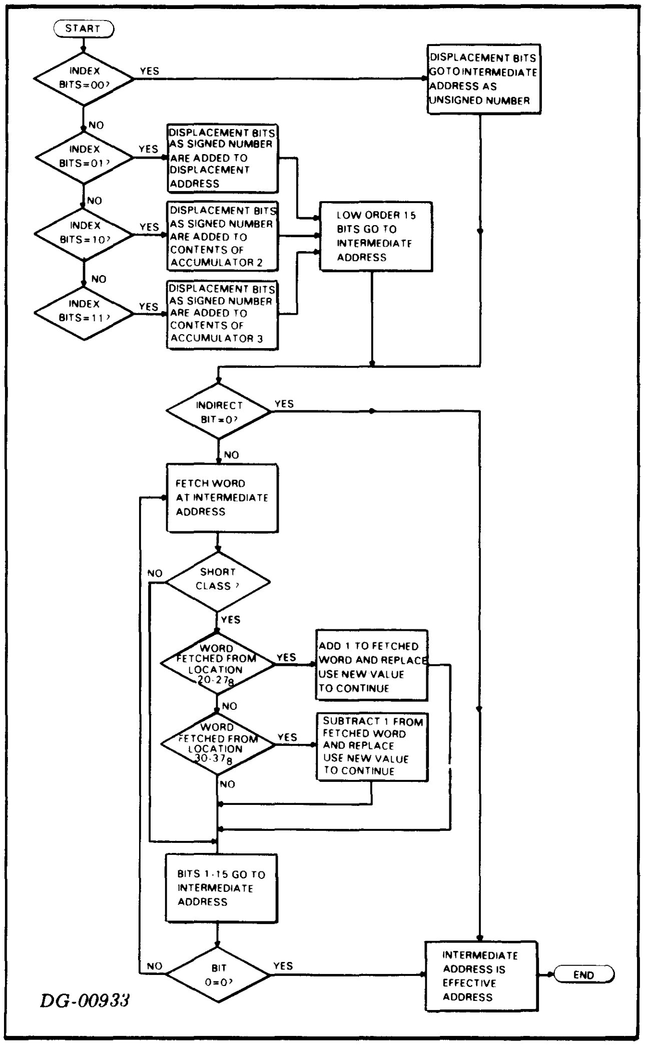
- The program may not utilize either the memory management and protection option or the hardware floating point option currently available for NOVA line computers.

- The memory and I/O resources available on the ECLIPSE series 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 series computers standard instruction set are functionally equivalent to the operations provided in the hardware multiply/divided option for the NOVA line computers. Only the operation codes must be changed to take advantage of the ECLIPSE series computer's multiply and divide feature. 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 floating point option. The floating point number formats are the same.

APPENDIX F
ADDRESSING

A flow diagram of the addressing process is shown below. See Chapter III for a detailed discussion of addressing.


START
INDEX BITS=00?
YES
DISPLACEMENT BITS GO TO INTERMEDIATE ADDRESS AS UNSIGNED NUMBER
NO
INDEX BITS=01?
YES
DISPLACEMENT BITS AS SIGNED NUMBER ARE ADDED TO DISPLACEMENT ADDRESS
NO
INDEX BITS=10?
YES
DISPLACEMENT BITS AS SIGNED NUMBER ARE ADDED TO CONTENTS OF ACCUMULATOR 2
LOW ORDER 15 BITS GO TO INTERMEDIATE ADDRESS
NO
INDEX BITS=11?
YES
DISPLACEMENT BITS AS SIGNED NUMBER ARE ADDED TO CONTENTS OF ACCUMULATOR 3
INDIRECT BIT=0?
YES
NO
FETCH WORD AT INTERMEDIATE ADDRESS
NO
SHORT CLASS?
YES
WORD FETCHED FROM LOCATION $20\text{-}27_8$
YES
ADD 1 TO FETCHED WORD AND REPLACE USE NEW VALUE TO CONTINUE
NO
WORD FETCHED FROM LOCATION $30\text{-}37_8$
YES
SUBTRACT 1 FROM FETCHED WORD AND REPLACE USE NEW VALUE TO CONTINUE
NO
BITS 1-15 GO TO INTERMEDIATE ADDRESS
NO
BIT 0=0?
YES
INTERMEDIATE ADDRESS IS EFFECTIVE ADDRESS
END

DG-00933

APPENDIX G
BOOTSTRAP LOADER

The *Program Load* console switch loads the bootstrap loader program shown below into the first 32_{10} words of memory and starts the program at location 0. See the console section of Chapter II for details on the use of the Program Load function.

```
BEG:      IORST                ;RESET ALL I/O
          READS     0          ;READ SWITCHES INTO ACO
          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 (AUTO-INCREMENT)
          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 ACO: (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
```

INSTRUCTION INDEX

* *EDIT Subinstruction*

* *EDIT Subinstruction*

I/O INSTRUCTION INDEX

BIBLIOGRAPHY

The following Data General publications may be of interest to readers of this manual:

Programmer's Reference, Peripherals	DGC No. 015-000021
Programmer's Reference, Data Control Unit	DGC No. 015-000060
Technical Reference, Data General Communications System	DGC No. 014-000070
Technical Manual, 6020 Series Tape Transport	DGC No. 015-000040
Technical Manual, Model 6045 6050 6051 Disc Drive (10 Megabyte)	DGC No. 015-000057
Technical Manual, DG/Disc Storage Subsystem (6060 Series, 100 Megabyte)	DGC No. 015-000061
Technical Manual, Model 6063-6065 Fixed Head Disc	DGC No. 015-000072
Interface Designer's Reference, NOVA and ECLIPSE Line Computers	DGC No. 015-000031
Software Summary and Bibliography	DGC No. 093-000110
AOS Software Documentation Guide	DGC No. 093-000202
AOS Programmer's Manual	DGC No. 093-000120
AOS Macroassembler Reference Manual	PGC No. 093-000192
AOS Binder User's Manual	DGC No. 093-000190
AOS Debugger and Disk File Editor User's Manual	DGC No. 093-000195
AOS System Manager's Guide	DGC No. 093-000193

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

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

Data General

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. 014-000616-01

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: __________

Data General

CUT ALONG DOTTED LINE

FOLD FOLD

STAPLE STAPLE

FOLD FOLD

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

Data General

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. 014-000600-01

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: __________

CUT ALONG DOTTED LINE

Data General

FOLD *FOLD*

STAPLE *STAPLE*

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

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. 014-000616-01

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: __________

Data General

CUT ALONG DOTTED LINE

FOLD FOLD

STAPLE STAPLE

FOLD FOLD

DataGeneral

ATTN: ENGINEERING PUBLICATIONS
4400 Computer Drive
Westboro, MA 01580

Data General

Data General Corporation, Westboro, Massachusetts 01580