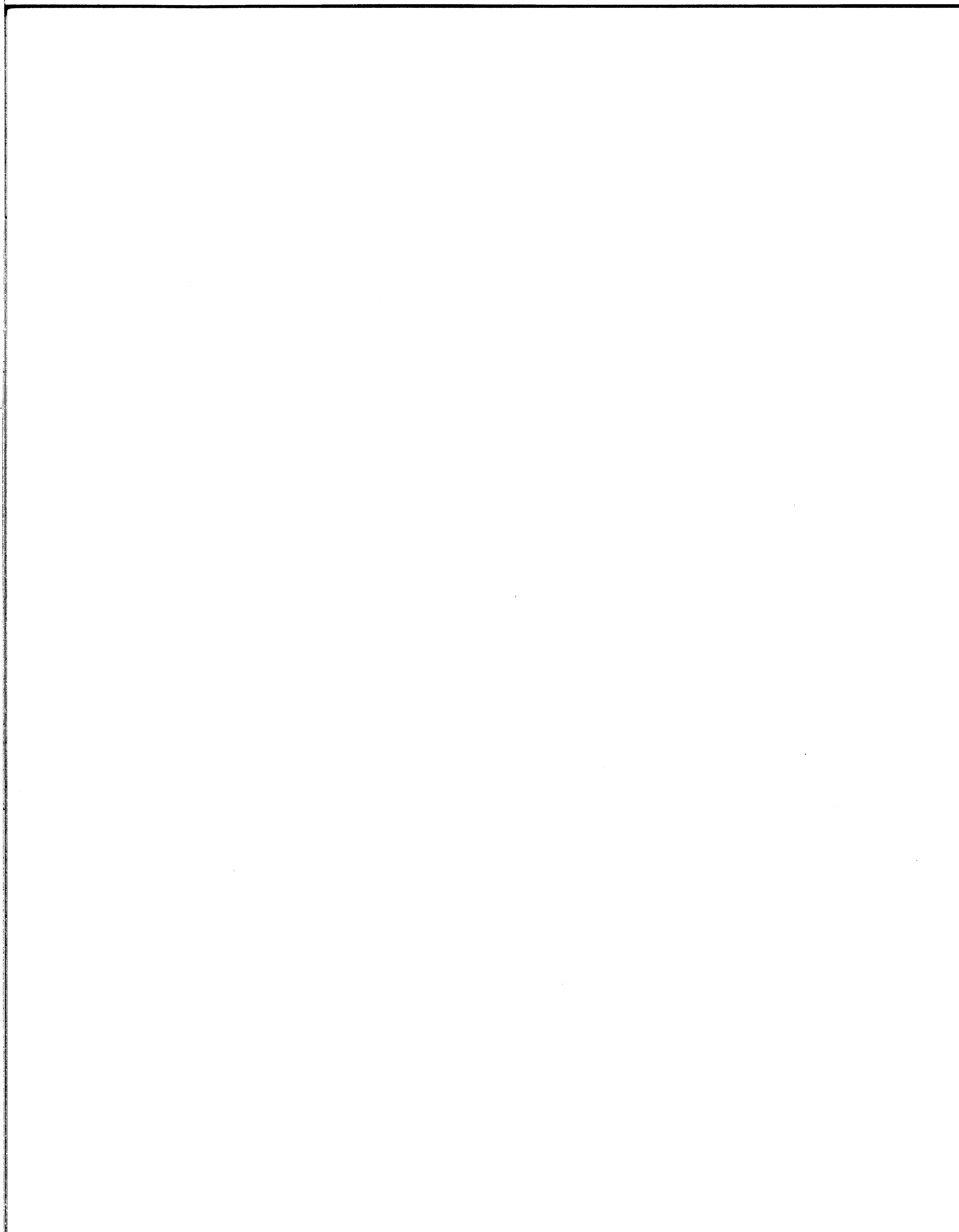
Programmer's Reference Series

# ECLIPSE® S/130

Data General



**Programmer's Reference Series**

# ECLIPSE® S/130

Data General Corporation, Westboro, Massachusetts 01581

# 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** and **microNOVA** are trademarks of Data General Corporation, Westboro, Massachusetts.

**FIRST EDITION**

*(First Printing, June 1979)*

Ordering No. 014-000621
© Data General Corporation, 1979
All Rights Reserved
Printed in the United States of America
Rev. 01, June 1979

# ECLIPSE® S/130

## PREFACE

*This manual describes the instruction set of the ECLIPSE® S/130 computer. We assume that our readers have a basic knowledge of assembly-language programming and of the binary, octal, and hex numbering systems. For a review of these numbering systems, see Appendix D of this manual.*

For a description of the assemblers available for the S/130, see the following manuals:

- *Absolute Assembler (DGC No. 093-000017)*
- *Extended Assembler (DGC No. 093-000040)*
- *Macro Assembler (DGC No. 093-000081)*
- *AOS Macro Assembler Reference Manual (DGC No. 093-000192)*

The following subjects are covered in this manual:

Chapter 1: Overview of the S/130; options available; software and peripheral support available.

Chapter 2: Data and addressing formats used.

Chapter 3: Description of the standard instruction sets.

Chapter 4: Description of the optional instruction sets, including the Floating Point, Character, and Map Instruction Sets.

Chapter 5: Description of the I/O instruction set including use of the interrupt system and *Vector* instruction.

Chapter 6: Use of the console.

Appendices

- Review of binary, octal and hexadecimal numbering systems.
- Summary of instruction execution times.
- Tables of I/O device codes.
- Tables of ASCII codes.

# TABLE OF CONTENTS


CHAPTER I — INTRODUCTION TO THE ECLIPSE S/130

I-1 INTRODUCTION
I-1 BASIC FEATURES
I-1 OPTIONAL FEATURES
I-2 SUPPORTING EQUIPMENT AND SOFTWARE
I-2 I/O Devices
I-2 Software
I-2 Other Documentation

CHAPTER II — DATA ÀND INSTRUCTION FORMATTING

II-1 INTRODUCTION
II-1 Bit Numbering Convention
II-2 DATA FORMATS
II-2 Integer Format
II-2 Floating Point Format
II-3 Sign
II-3 Exponent
II-3 Mantissa
II-3 Logical Format
II-3 Byte Format
II-3 Decimal Format
II-4 ADDRESSING CONVENTIONS
II-4 Word Addressing
II-5 Addressing Modes
II-5 Absolute Addressing Mode
II-5 P.C. Relative Addressing Mode
II-5 Accumulator Relative Addressing Mode
II-5 Direct and Indirect Addressing
II-6 Auto-Incrementing and -Decrementing
II-7 BYTE ADDRESSING
II-7 BIT ADDRESSING
II-8 ADDRESSING WITH THE MAP

## TABLE OF CONTENTS (Continued)


| | |
|---|---|
| II-8 | RESERVED STORAGE LOCATIONS |
| II-9 | PROGRAM EXECUTION |
| II-9 | Sequential Operation |
| II-9 | Program Flow Alteration |
| II-9 | Program Flow Interruption |
| II-10 | INSTRUCTION FORMAT |
| II-10 | Memory Address Formats |
| II-10 | Immediate Formats |
| II-10 | Two-Accumulator-Multiple Operation Format |
| II-12 | I/O Instruction Format |
| CHAPTER III | INSTRUCTION SETS |
| III-1 | INTRODUCTION |
| III-1 | CODING AIDS |
| III-3 | FIXED POINT ARITHMETIC |
| III-8 | Programming Examples |
| III-8 | Two-Accumulator-Multiple Operation Instructions |
| III-8 | Double-Precision Arithmetic |
| III-8 | Double-Precision Addition |
| III-8 | Double-Precision Subtraction |
| III-9 | LOGICAL OPERATIONS |
| III-12 | Example |
| III-13 | CONDITIONAL INSTRUCTION SET |
| III-15 | Programming Examples |
| III-15 | BYTE MANIPULATION |
| III-16 | Programming Example |
| III-16 | BIT MANIPULATION |
| III-18 | Programming Example |
| III-19 | BLOCK MANIPULATION |
| III-20 | Programming Example |
| III-20 | THE STACK |
| III-20 | Introduction |
| III-21 | Stack Control Words |
| III-21 | Stack Pointer |
| III-21 | Stack Limit |
| III-21 | Stack Fault Address |
| III-21 | Frame Pointer |

## TABLE OF CONTENTS (Continued)


III-22 Stack Protection
III-22 Stack Overflow
III-22 Stack Underflow
III-23 Stack Protection Fault
III-23 Stack Overflow Protection
III-23 Stack Underflow Protection
III-23 Stack Fault Handler
III-24 Initializing the Stack Control Words
III-24 Stack Pointer
III-24 Stack Limit
III-24 Stack Fault Address
III-24 Frame Pointer
III-24 Examples
III-25 STACK INSTRUCTIONS
III-27 Programming Example
III-28 PROGRAM FLOW ALTERATION
III-30 SUBROUTINE CALLS AND RETURNS
III-30 Introduction
III-30 Properties of Subroutine Calls and Routines
III-30 Programming Examples
III-30 Example 1
III-31 Example 2
III-31 Example 3
III-32 EXTENDED OPERATION FEATURE

CHAPTER IV OPTIONAL FEATURES OF THE ECLIPSE S/130

IV-1 INTRODUCTION
IV-1 MEMORY ALLOCATION AND PROTECTION
IV-2 Address Translation
IV-2 Sharing of Physical Memory
IV-3 Types of Maps
IV-3 Supervisor Mode
IV-3 MAP Protection Capabilities
IV-3 Validity Protection
IV-3 Write Protection
IV-3 Indirection Protection
IV-4 I/O Protection

## TABLE OF CONTENTS (Continued)


| Page | Section |
|---|---|
| IV-4 | MAP Protection Faults |
| IV-4 | Load Effective Address Mode |
| IV-4 | Initial Conditions |
| IV-5 | MAP INSTRUCTIONS |
| IV-9 | FLOATING POINT ARITHMETIC |
| IV-9 | Floating Point Registers |
| IV-9 | Guard Digit |
| IV-10 | Floating Point Fault Conditions |
| IV-10 | Floating Point Trap |
| IV-19 | CHARACTER INSTRUCTION SET |
| IV-24 | Programming Examples |
| IV-25 | WRITABLE CONTROL STORE |
| IV-25 | Placing Microcode In WCS |
| CHAPTER V | INPUT/OUTPUT |
| V-1 | INTRODUCTION |
| V-1 | THE S/130 I/O SYSTEM |
| V-1 | Programmed I/O |
| V-2 | Data Channel I/O |
| V-2 | Device Codes |
| V-2 | Busy and Done Flags |
| V-2 | Data Channel |
| V-3 | I/O INSTRUCTIONS |
| V-4 | Programming Example |
| V-5 | I/O Interrupts |
| V-5 | Interrupt System Definitions |
| V-5 | Processing an Interrupt |
| V-6 | Priority Interrupt System |
| V-6 | Setting Up a Priority System |
| V-7 | Priority Interrupt Handler |
| V-7 | Stack Changes |
| V-8 | INTERRUPT INSTRUCTIONS |
| V-9 | SPECIAL CENTRAL PROCESSOR INSTRUCTIONS |
| V-10 | USE OF THE VECTOR INSTRUCTION |
| V-12 | Common Process |
| V-12 | Mode A |

## TABLE OF CONTENTS (Continued)


| Page | Entry |
|---|---|
| V-12 | Mode B |
| V-12 | Mode B - Part I |
| V-12 | Mode C - Part I |
| V-12 | Mode D - Part I |
| V-13 | Mode E - Part I |
| V-13 | Mode B through E - Part II |
| V-13 | Programming Example |
| V-14 | ERROR CHECKING AND CORRECTION |
| V-14 | Method of Operation |
| V-15 | REAL TIME CLOCK |
| V-16 | POWER FAIL/AUTO-RESTART |
| V-16 | BATTERY BACKUP |
| CHAPTER VI | CONSOLE |
| VI-1 | INTRODUCTION |
| VI-1 | CONSOLE SWITCHES |
| VI-2 | Reset-Stop |
| VI-2 | Deposit Examine |
| VI-2 | Exam-Exam Nxt |
| VI-2 | Inst-Micro/Inst |
| VI-2 | PR-Load-Exec |
| VI-3 | Start-Cont |
| VI-3 | Dep-Dep Nxt |
| VI-3 | Address Compare |
| VI-3 | Off |
| VI-3 | Monitor |
| VI-3 | Stop/Store |
| VI-3 | Stop/Addr |
| VI-3 | Power |
| VI-4 | PROGRAM LOADING |
| VI-5 | Self Test Function |

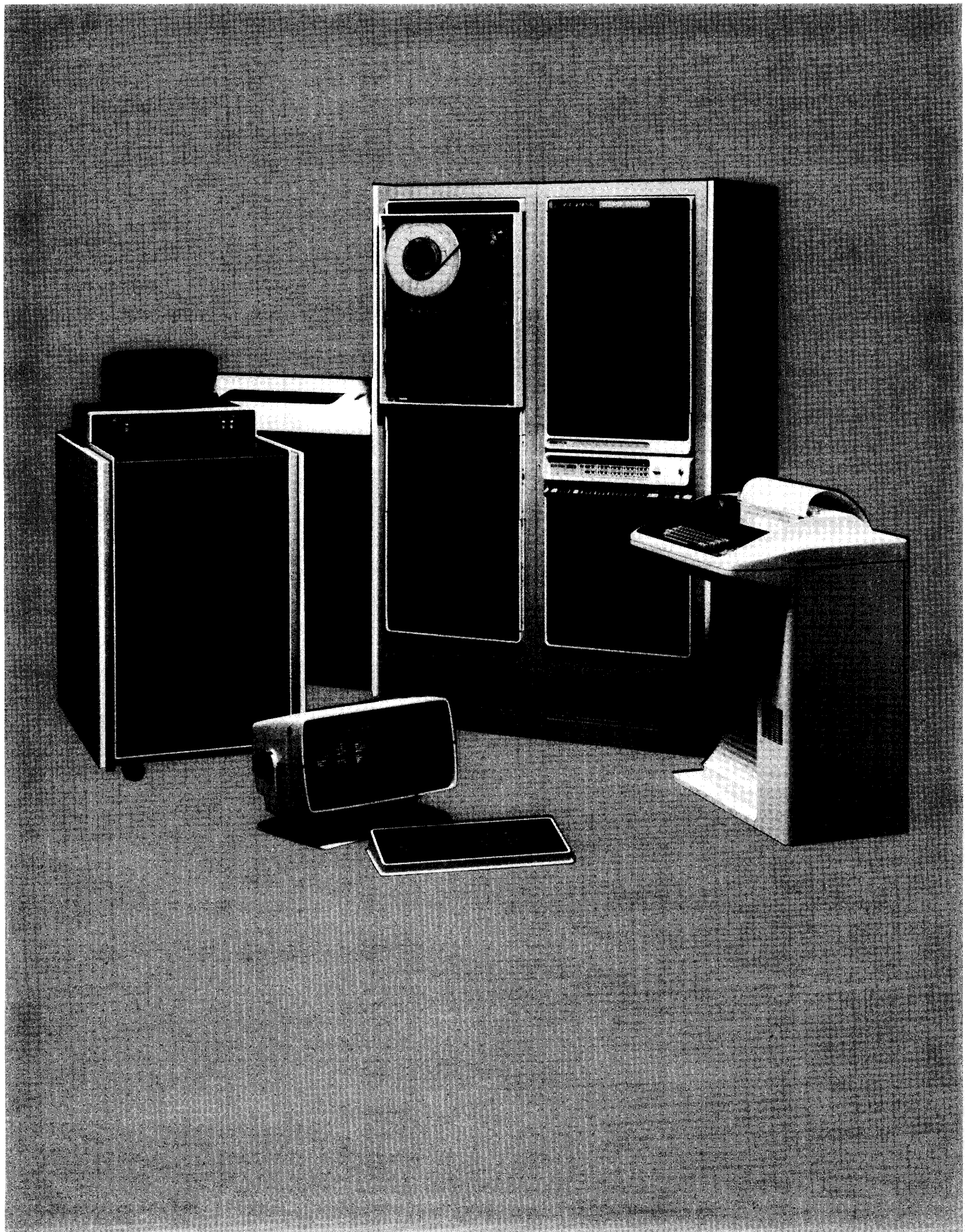
## TABLE OF CONTENTS (Continued)

APPENDIX A STANDARD I/O DEVICE CODES
APPENDIX B OCTAL AND HEXADECIMAL CONVERSION
APPENDIX C ASCII CHARACTER CODES
APPENDIX D BINARY, OCTAL AND DECIMAL NUMBERING SYSTEMS
APPENDIX E COMPATIBILITY WITH NOVA LINE COMPUTERS
APPENDIX F INSTRUCTION EXECUTION TIMES
APPENDIX G INSTRUCTION USE EXAMPLES

INSTRUCTION INDEX

BIBLIOGRAPHY

This page intentionally left blank.



# CHAPTER I
# INTRODUCTION TO THE ECLIPSE® S/130

## INTRODUCTION

Data General Corporation's ECLIPSE S/130 computers are general purpose computers intended for use in all types of system applications, including instrumentation and control, communications, and data processing. The important features of the S/130 are summarized below, and covered in detail in the chapters that follow.

## BASIC FEATURES

The basic ECLIPSE S/130 has the following features:

- Word length of 16 bits (two 8-bit bytes).
- Four 16-bit accumulators.
- Direct and indirect addressing of 32,768 words (64K bytes) of memory.
- Both programmed and data channel I/O for efficient data transfer between main memory and peripheral devices.
- Priority interrupt handling, with vectoring capability for automatic dispatch to the correct interrupt handler.
- Main-memory resident push-down stacks for temporary information storage, with a separately definable stack areas for the operating system, each user, and the priority interrupt handler.
- Asynchronous memory and device operation for maximum speed and flexibility.
- Microprogrammed instruction sets.
- Control store parity checking.

## OPTIONAL FEATURES

In addition to the above, the following optional features are also available:

- Memory Allocation and Protection (MAP) which increases usable memory to 131,072 16-bit words (256K bytes).
- Floating point instruction set providing up to 17 significant decimal digits within a range of approximately $5.4 \times 10^{-79}$ to $7.2 \times 10^{+75}$ Four 64-bit accumulators are provided for floating point operations.
- Character instruction set for increased flexibility when manipulating character strings.
- Error Checking and Correction (ERCC) to detect and correct single-bit memory errors.
- Writeable Control Store (WCS) providing 1024 microstore locations for storing user-created microinstructions.
- User Control Store (UCS) providing 2048 microstore locations for a user-specified custom instruction set.
- Battery backup for preserving semiconductor memory during short power failures.
- Automatic power-fail/auto-restart capability.

## SUPPORTING EQUIPMENT AND SOFTWARE

The usefulness of the S/130 is increased by the variety of I/O devices and software available for it. Below is a summary of available I/O devices and software.

### I/O Devices

- Teletypewriters, video displays, line printers, and real-time clock.
- Paper tape, magnetic tape and disc storage units.
- Multiplexors and telecommunications adapters.
- Synchronous and asynchronous interfaces, including an IBM 360/370 interface.

### Software

- The Stand-Alone Operating System (SOS), Real Time Operating System (RTOS), Real Time Disk Operating System (RDOS) and Mapped RDOS for systems requiring one or two users.
- The Advanced Operating System (AOS) for systems requiring more than two users.
- An assembler and macro assembler for translating assembly-language mnemonics into machine-language binary representation.
- A microcode assembler and loader for use with the WCS feature.
- Software to support higher-level languages such as ALGOL, EXTENDED BASIC, and FORTRAN5.

### Other Documentation

Below is a list of other documentation you might find useful:

- *S/130 Technical Manual (DGC. No. 015-000070)*
- *Interface Designer's Reference (DGC No. 015-000031)*
- *Programmer's Reference, Peripherals (DGC No. 015-000021)*
- *RDOS Reference Manual, (DGC No. 093-000075)*
- *Introduction to RDOS, (DGC No. 093-000083)*
- *Learning to Use Your RDOS, (DGC No. 093-000223)*
- *AOS Programmer's Manual, (DGC No. 093-000120)*
- *Introduction to AOS, (DGC No. 093-000121)*
- *AOS Microassembler Reference Manual, (DGC No. 093-000192)*
- *Learning to Use Your AOS, (DGC No. 093-000196)*
- *AOS Software Documentation Guide (DGC No. 093-000202)*

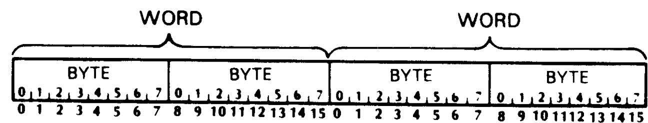
# CHAPTER II
# DATA AND INSTRUCTION FORMATTING

## INTRODUCTION

The formatting conventions used for entering data and instructions in the S/130 are described in this chapter. Some of the conventions used require a knowledge of the binary, octal, and hexadecimal numbering systems. See Appendix D for a review of these numbering systems.

### Bit Numbering Convention

In the S/130, bits contained in bytes and words are numbered from left to right, with the leftmost (high-order) bit always numbered 0. Numbering of bits is always done in the decimal numbering system.



# DATA FORMATS

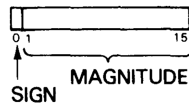
## Integer Format

We represent a signed integer by a twos-complement number in one or more 16-bit words, with bit 0 of the first word representing the sign and the remaining bits representing the magnitude. The sign of the number is positive if the sign bit is 0 and negative if the sign bit is 1.

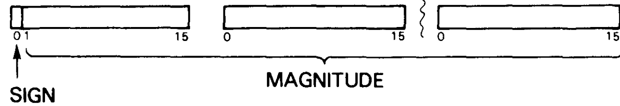
We represent an unsigned integer by a binary quantity using all the bits of one or more 16-bit words to represent the magnitude.

### SIGNED INTEGERS

SINGLE PRECISION:

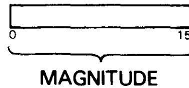



MULTIPLE PRECISION:




### UNSIGNED INTEGERS

SINGLE PRECISION:




MULTIPLE PRECISION:

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, a *carry* bit is provided. A change in the value of the carry bit indicates a carry out during an add or subtract operation.

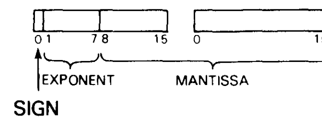
## Floating Point Format

Floating point format provides a much larger range than integer format, at the expense of some precision. It also provides the capability 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.

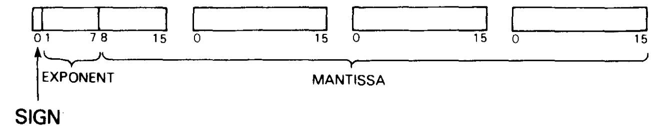
The floating point number is represented by

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

SINGLE PRECISION:




DOUBLE PRECISION:




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

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

Zero is represented by a floating point 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 word 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

Bits 1-7 contain the exponent. We use *excess 64* representation in the exponent field to obtain both positive and negative exponents. In this representation, the value in the exponent field is 64 greater than the true value of the exponent. This is illustrated below:

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

### Mantissa

Bits 8-31 (single precision) or bits 8-63 (double precision) contain the mantissa. The binary point is assumed to be to the left of bit 0 of the mantissa (bit 8 of the whole 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 represent a nonzero quantity. For every hex digit shifted, the exponent is decreased by 1. Since the mantissa is shifted 4 bits at a time, it is possible for the high-order three bits of a normalized mantissa to be zero.

## Logical Format

Logical format is used when manipulating logical entities. Each bit of a 16-bit word is treated as a separate logical entity. When two words are involved (logical AND or XOR, for example) only equal-numbered 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.

## Byte Format

Byte format is used when manipulating bytes, particularly alphanumeric characters. Each 8-bit byte is treated as an unsigned binary integer and is selected by a 16-bit *byte pointer* (See *Addressing,* below).

## Decimal Format

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

# ADDRESSING CONVENTIONS

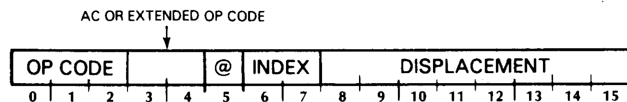
The various methods of addressing memory locations in the S/130 give the programmer considerable flexibility when storing and retrieving data, or tranferring control to a different procedure.

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. The memory location after location $77777_8$ is location 0.

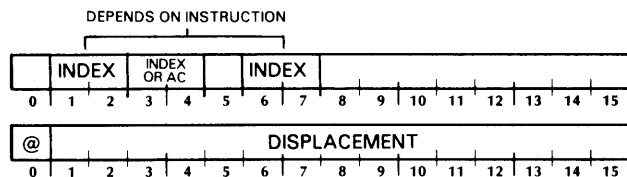
## Word Addressing

The following definitions are useful for understanding word addressing in the S/130:

SHORT CLASS:




EXTENDED CLASS:




**Addressing Modes** - Three methods of addressing using a displacement from some reference point to find the desired address. Different modes use 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.

**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. Below 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 $32768_{10}$ |
| 01 10 11 | -200 to $+177_8$ or -128 to $+127_{10}$ | $-40000_8$ to $37777_8$ or $-16{,}384_{10}$ to $+16{,}383_{10}$ |

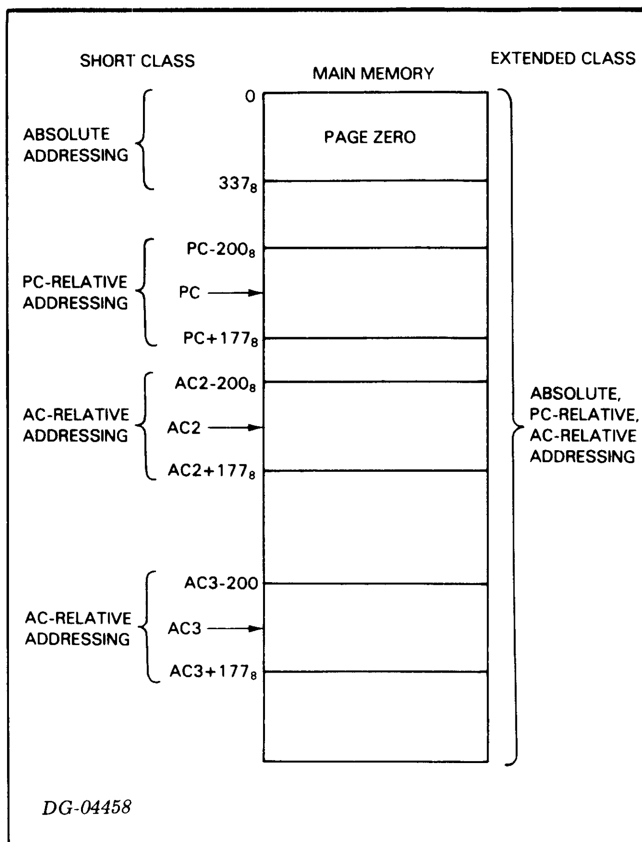
## Addressing Modes

Word addressing in the S/130 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.




*DG-04458*

## Absolute Addressing Mode

In absolute addressing mode, the effective address is set equal to the unmodified displacement. As a result, the short class of instructions can only address locations in the range 0-$377_8$ in the absolute mode (short class instructions are restricted to 8 bits in the displacement).

Page zero thus becomes very important because any memory-reference instruction can address any word in this page. It is used as a common storage area for constants that are frequently referenced throughout the program.

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

## P.C. Relative Addressing Mode

In P.C. relative addressing mode, the effective address is found by adding the displacement to the contents of the program counter (i.e., to the address of the memory-reference instruction itself).

## Accumulator Relative Addressing Mode

In accumulator relative addressing mode, the effective 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 also is used as a pointer 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 S/130, but in mapped systems, indirect protection is available which can limit indirections to 15 levels (see the MAP section, Chapter IV).

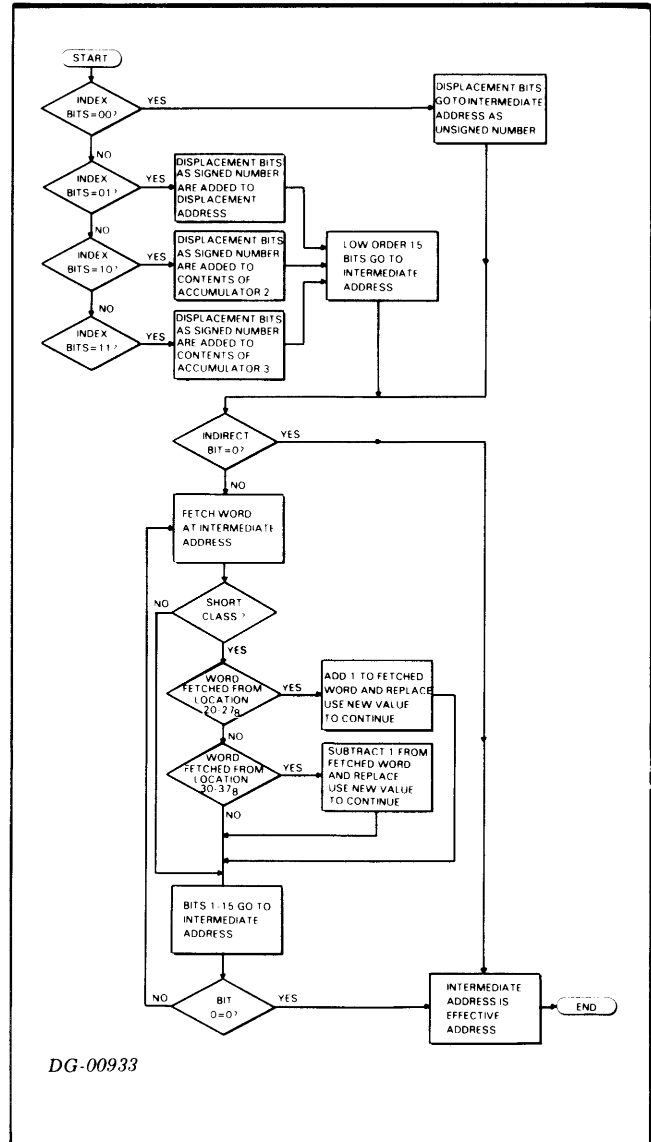
## Auto-Incrementing and -Decrementing

If the intermediate address of a short class instruction is in the range $20\text{-}27_8$, and the indirect bit is 1, the contents of the addressed location are incremented by one. The incremented value is used to continue the addressing chain.

If the intermediate address of a short class instruction is in the range $30\text{-}37_8$, and the indirect bit is 1, the contents of the addressed location are decremented by one. The decremented value is used to continue the addressing chain.

> **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.*

Below is a flow diagram of the addressing process.

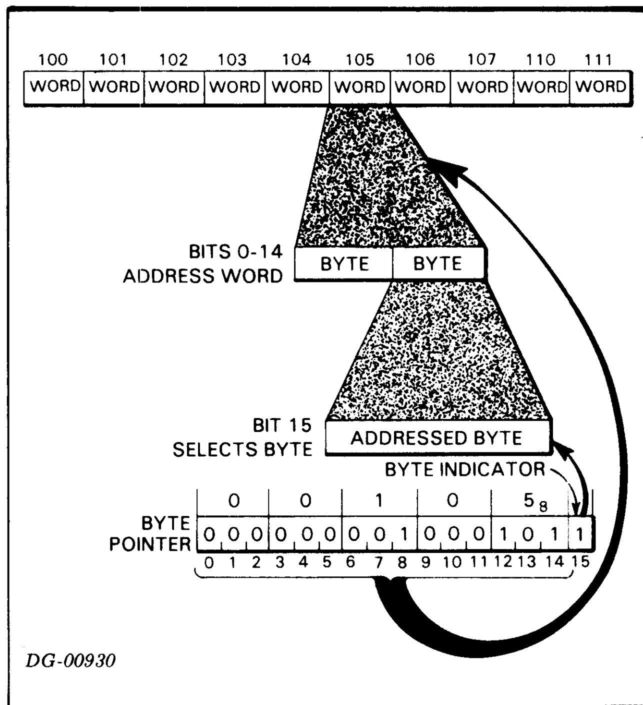



DG-00933

## BYTE ADDRESSING

There are eight instructions that use byte addressing in the S/130. Six of these are the instructions in the character instruction set.

These instructions address bytes using 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

## BIT ADDRESSING

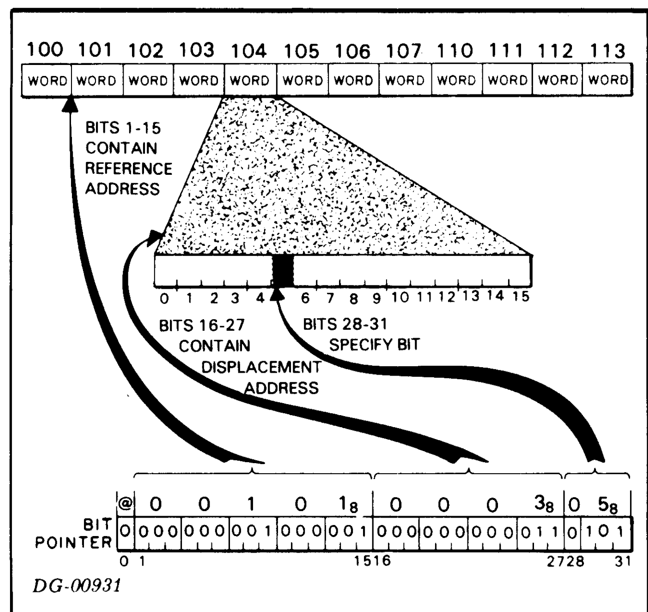
There are five instructions that use bit addressing in the S/130. These instructions address bits using a 32-bit (2-word) *bit pointer.*

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.

The address of the desired bit is obtained as follows:

The address formed by the positive number contained in bits 1-15 of the bit pointer (the *reference* address) is added to the address formed by the 12-bit positive number contained in bits 16-27 (the *displacement* address). 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. None of this computation affects the original contents of the bit pointer.

Below is a diagram of the bit-addressing process.



DG-00931

## ADDRESSING WITH THE MAP

The MAP increases the amount of usable *physical* address space, making multi-user systems practical since each of several users can then have a full 32K of memory if that is required. Installation of a MAP has no effect on addressing from the programmer's point of view, unless the MAP itself is being programmed. The maximum amount of *logical* address space available to the programmer is 32,768 words.

The MAP intercepts memory references and translates the 15-bit logical address into a 17-bit physical address. The translation functions are programmed into the MAP, but the translation process is invisible to the regular user.

**NOTE** *For those unmapped systems with less than 32K, the maximum useful address space is equal to the amount of memory available.*

## RESERVED STORAGE LOCATIONS

There are 32 reserved storage locations in the S/130. 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.

Note that the first 4 locations in the table must be physical locations. The next 4 could be physical or logical, depending on whether the *Vector* instruction is executed in mapped mode or not (generally it is not). The last 8 are all logical locations.

| LOCATION ADDRESS (Octal) | LOCATION NAME | LOCATION FUNCTION |
|---|---|---|
| 0 | I/O RETURN ADDRESS | Return address from I/O interrupt. Also first instruction of Auto-restart routine. |
| 1 | I/O HANDLER ADDRESS | Address of the I/O interrupt handler. Indirectable. |
| 2 | SC HANDLER ADDRESS | Address of the SYSTEM CALL instruction handler. Indirectable. |
| 3 | PF HANDLER ADDRESS | Address of the protection fault handler. Indirectable. |
| 4 | VECTOR STACK POINTER | Address of the top of the VECTOR STACK. Non-indirectable. |
| 5 | CURRENT MASK | Current interrupt priority mask. |
| 6 | VECTOR STACK LIMIT | Address of the last normaly usable location in the VECTOR stack. |
| 7 | VECTOR STACK FAULT ADDRESS | Address of the VECTOR stack fault handler. Indirectable. |
| 20-27 | AUTO-INC0 through AUTO-INC7 | Auto-incrementing locations. |
| 30-37 | AUTO-DEC0 through AUTO-DEC7 | Auto-decrementing locations. |
| 40 | STACK POINTER | Address of the top of the stack. Non-indirectable. |
| 41 | FRAME POINTER | Address of the start of the current stack frame minus 1. Non-indirectable. |
| 42 | STACK LIMIT | Address of the last normally usable location in the stack. |
| 43 | STACK FAULT ADDRESS | Address of the stack fault handler. Indirectable. |
| 44 | XOP ORIGIN ADDRESS | Address of the beginning of XOP table. Non-indirectable. |
| 45 | FLOATING POINT FAULT ADDRESS | Address of the floating point fault handler. Indirectable. |
| 46 | | Reserved for future use. |
| 47 | | Reserved for future use. |

# PROGRAM EXECUTION

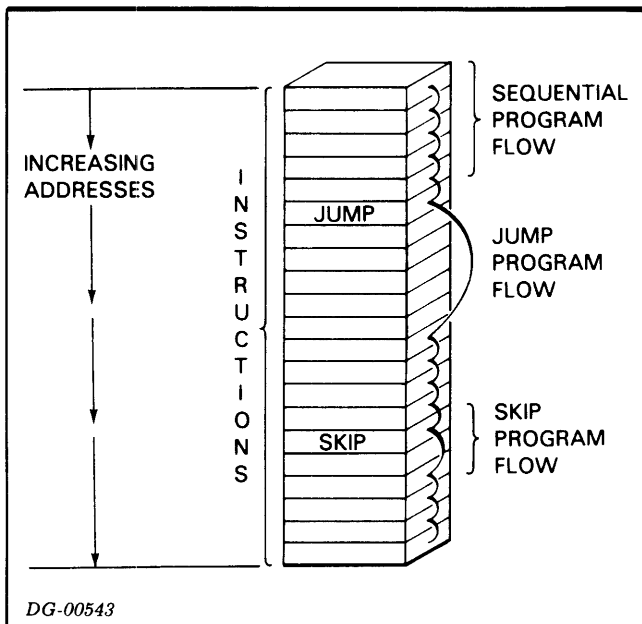
## Sequential Operation

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 address the complete logical address space, i.e., $77777_8$ ($32{,}768_{10}$) 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

The program flow can be altered from sequential operation by the programmer in two ways. Jump instructions alter the program flow by inserting a new value into the program counter. Conditional skip instructions can 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.

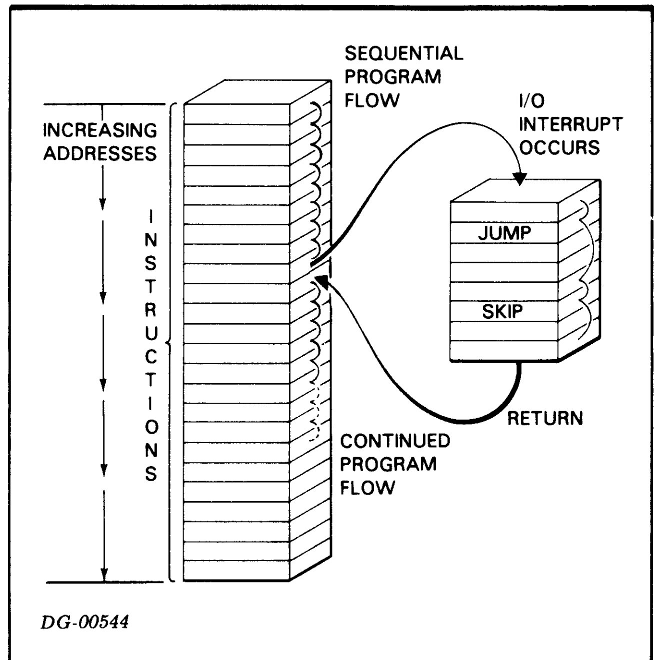
**NOTE** *Do not use a conditional skip immediately before an extended class (2-word) instructions. Doing so will result in an attempt to execute the second word of the instruction as a short class instruction.*



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 various kinds of 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

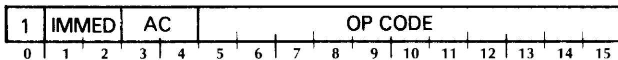
# INSTRUCTION FORMAT

The format for each instruction is shown at the instruction description. Most of the formats are quite simple and need no additional explanation. Those that do are discussed in this section.

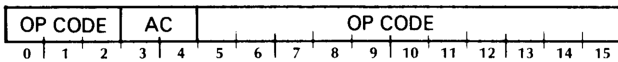
Below are diagrams of the formats which are discussed below.

IMMEDIATE FORMAT

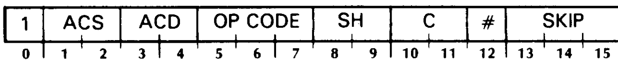
SHORT CLASS:



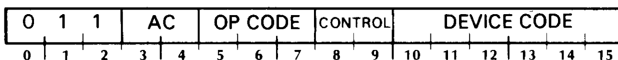

EXTENDED CLASS:



TWO ACCUMULATOR-MULTIPLE OPERATION FORMAT




I/O FORMAT




## Memory Address Formats

There are four types of memory address formats - two with and two without an accumulator reference. The memory address portion of the format is identical, however, and was discussed in detail in the section on Addressing Conventions. When applicable, include the identity of the accumulator as indicated in the format diagrams.

## Immediate Formats

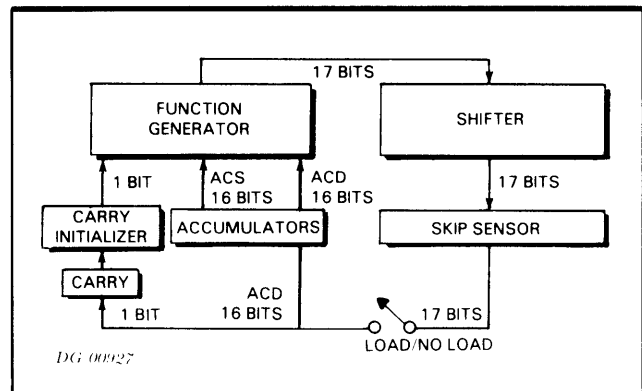
When a constant is needed only once or twice in a program, it is more efficient to obtain it by using an *immediate* rather than retrieving it from memory. An immediate is a constant which is stored in the instruction word itself, or in the next memory word, and is coded along with the instruction.

The short class of immediate instructions codes the value of the immediate minus 1 into two bits of the instruction word. This permits immediates in the range 1-4 using just 2 bits. When the instruction is executed, the value in the immediate field is incremented by 1 before it is used.

The extended class of immediate instruction codes the value of the immediate into the second instruction word with no change.

## Two Accumulator-Multiple Operation Format

The two accumulator-multiple operation instructions use an arithmetic unit which can perform several operations on two data words, using one instruction. The logical organization of the arithmetic unit is illustrated below.



The function generator is capable of providing 8 functions: *Add, Subtract, Negate, Add Complement, Move, Increment, Complement,* and *AND.* Bits 5, 6, and 7 contain the operation code which controls the function generator.

All the instructions using this arithmetic unit operate on the contents of 1 or 2 accumulators and the carry bit. Bits 1 and 2 specify the accumulator containing the source operand, and bits 3 and 4 specify the accumulator containing the destination operand. Bits 10 and 11 specify the initial value of the carry bit.

These instructions specify a shift operation, which is performed by the shifter in the arithmetic unit. The shifter operates on the 17-bit quantity consisting of the 16-bit output of the ALU and the carry bit. The shifter can rotate this quantity left or right, or swap the two bytes in the accumulator. Bits 8 and 9 control the operation of the shifter.

Bits 13-15 specify the skip test. The arithmetic unit can test the result of the function generator/shifter combination for various conditions, and skip or not skip the next instruction depending on the results of the test.

Bit 12 is the no-load bit. If this bit is 1, the results of the shift operation are not loaded into the destination accumulator, but all the other operations (such as skip tests) take place.

> **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 table below summarizes the various operations that can be performed by the two accumulator-multiple operation instruction. The characters in the column title *Format Designator* refers to the designator used in the format diagram. The *Coded Character* is the character recognized by the assembler that produces the *Result Bits* shown in the next column.

| FORMAT DESIG. | CODED CHARACTER | RESULT BITS | OPERATION |
|---|---|---|---|
| C | (option omitted) | 00 | Do not initialize the carry bit |
| | Z | 01 | Initialize the carry bit to 0 |
| | O | 10 | Initialize the carry bit to 1 |
| | C | 11 | Initialize the carry bit to the complement of its present value |
| SH | (option omitted) | 00 | Leave the result of the arithmetic or logical operation unaffected |
| | L | 01 | Combine the carry and the 16-bit result into a 17-bit number and rotate it one bit left |
| | R | 10 | Combine the carry and the 16-bit result into a 17-bit number and rotate it one bit right |
| | S | 11 | Exchange the two 8-bit halves of the 16-bit result without affecting the carry |
| # | (option omitted) | 0 | Load the result of the shift operation into ACD |
| | # | 1 | Do not load the result of the shift operation into ACD and restore carry to its original state |
| SKIP | (option omitted) | 000 | Never skip |
| | SKP | 001 | Always skip |
| | SZC | 010 | Skip if carry = 0 |
| | SNC | 011 | Skip if carry ≠ 0 |
| | SZR | 100 | Skip if result = 0 |
| | SNR | 101 | Skip if result ≠ 0 |
| | SEZ | 110 | Skip if either carry or result = 0 |
| | SBN | 111 | Skip if both carry and result ≠ 0 |

The following diagrams illustrate the operation of the shifter.

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

## I/O Instruction Format

The I/O instructions control the transfer of data between the computer and the various I/O devices. This is done through the manipulation of a Busy and Done flag in each device (see Chapter V). Depending on the instruction, bits 8 and 9 contain a code which either changes the value of the flags or tests one of them for a certain value.

The tables below summarize the operations performed by the optional mnemonics used in I/O instructions. The first table applies to those instructions which change the Busy and Done flags.

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

The next table applies to the I/O SKIP instruction which tests the Busy and Done flags.

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

The last table applies to I/O instructions with a device code of $77_8$. These instructions are used by the Interrupt System and for special CPU functions. Instead of manipulating or testing the Busy and Done flags, these instructions operate on the Interrupt On and Power Fail flags.

| CLASS ABBREV. | CODED CHARACTER | RESULT BITS | OPERATION |
|---|---|---|---|
| f | (option omitted) | 00 | Does not affect the state of the Interrupt On flag |
| | S | 01 | Set the Interrupt On flag to 1 |
| | C | 10 | Set the Interrupt On flag to 0 |
| | P | 11 | Does not affect the state of the Interrupt On flag (Used only in the VCT instruction) |
| t | BN | 00 | Test for Interrupt On = 1 |
| | BZ | 01 | Tests for Interrupt On = 0 |
| | DN | 10 | Tests for Power Fail = 1 |
| | DZ | 11 | Tests for Power Fail = 0 |

Bits 10-15 contain the *device code* which identifies the I/O device being addressed by the instruction. In some cases, bits 3 and 4 identify an accumulator used in the data transfer.

# CHAPTER III
# INSTRUCTION SETS

## INTRODUCTION

This chapter contains a complete description of the machine-level instructions in the S/130, including:

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

In addition, examples are included to clarify the function of the instruction.

## CODING AIDS

Various conventions and abbreviations are used throughout this manual to aid the programmer in properly coding each instruction. Their definitions are given below:

[ ], *[ ]* — Square brackets indicate that the enclosed symbol (e.g., *[,skip]*) is an optional operand or mnemonic, and may be coded or not, depending upon whether or not the associated option is desired.

**BOLD** — Operands or mnemonics printed in boldface must be coded exactly as shown. For example, the mnemonic for the *Move* instruction is coded **MOV**.

*italic* — Operands or mnemonics printed in italics require a specific substitution. Replace with a number or symbol that provides the assembler with the proper accumulator number, user-defined symbol, address, etc., as appropriate.

The following abbreviations are used throughout this manual:

| | | |
|---|---|---|
| I | = | Either signed two's complement integer in the range -32,768 to +32,767 or unsigned integer in the range 0 to +65,535. |
| N | = | Integer in the range 0-3 |
| n | = | Integer in the range 1-4 |
| AC | = | Accumulator |
| ACS | = | Source Accumulator |
| ACD | = | Destination Accumulator |
| FPAC | = | Floating Point Accumulator |
| FACS | = | Floating Point Source Accumulator |
| FACD | = | Floating Point Destination Accumulator |

## ASSEMBLER CONVENTIONS

Some of the more important relationships between assembler and machine instructions are discussed below. For a detailed discussion of the assembler, see one of the assembler manuals listed in Chapter I.

### RELOCATABILITY

The *Relocatable Loader (DGC No. 093-000130)* uses the concept of relocatability to make memory use more efficient. Code which does not have to be in some particular memory location is so designated by the assembler, permitting the loader to place it wherever is most convenient. Most code is in this catagory. An example of code which must be placed at a particular location is the code defining the stack.

### PSEUDO-OPS

The assembler expects to find certain expressions in the user program which control the operation of the assembler, but do not directly control the processing of the program after assembly. These expressions are called *pseudo-ops.* An example of a pseudo-op is .END which marks the end of a source program.

### PAGE ZERO

Extended class memory reference instructions can address any one of 32,768 words (i.e., all of the maximum logical address space) in memory using absolute addressing (see Chapter 2, *Instruction Formats).* The short class of instruction, however, has only 8 bits in the displacement field, and therefore can only address locations 0-$377_8$ (0-$255_{10}$) using absolute addressing. The first 256 words of memory are therefore given a special status, because they are accessible by any memory reference instruction anywhere in the program, using absolute addressing. The assembler manuals refer to these first 256 locations as *page zero.*

Variables can be stored in relocatable form in either page zero, or in the rest of memory. The assembler uses the pseudo-op .ZREL to indicate that the lines following (until otherwise indicated) are to be assembled in relocatable form in page zero. The pseudo-op .NREL indicates relocatable code outside of page zero and .ABS indicates nonrelocatable code placed in the specified location.

### RADIX

The assembler assumes that all numbers in the source program are in the numbering system corresponding to the current *radix.* The default radix is base 8, or octal. Decimal, hexadecimal and binary are other commonly used radices. The assembler radix can be changed using the .RDX pseudo-op.

### MEMORY REFERENCE INSTRUCTIONS

In the instructions which calculate an effective address, the assembler expects the following coding conventions to be used:

*Coding the symbol @ anywhere in the effective address operand string sets the indirect bit to 1.*

*Coding a comma followed by one of digits 0-3 as the last operand of the operand string sets that value in the index field. Use the symbol* dot *(.) to set the index bits to 01.* Dot *can be read to mean* address of the instruction. *When the dot is used, it is followed by either a plus or minus sign followed by the displacement, e.g.,* .+7, *or* .-2.

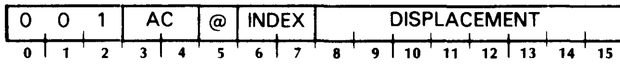
**NOTE** *When coding extended class instructions, setting the index bits to 01 using the dot (e.g.,* EJMP .+d *) does not produce the same effect as coding a comma followed by a 1* (EJMP d,1)*. When using the dot, the displacement is added to the address of the instruction (the first word of a 2-word instruction). When using the comma, the displacement is added to the address of the word containing the displacement (the second word of a 2-word instruction). Therefore,* EJMP .+3 *is equivalent to* EJMP 2,1.

# FIXED POINT ARITHMETIC

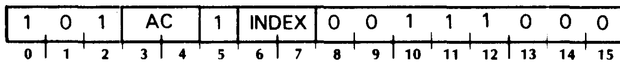
The fixed point instruction set performs binary arithmetic on operands in accumulators. The operands are 4, 16, or 32 bits in length and can be either signed or unsigned.

## Load Accumulator

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



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

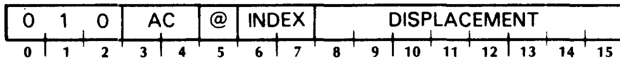


Moves a word out of memory and into an accumulator.

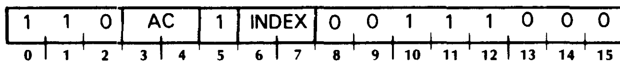
The word addressed by the effective address, *E*, is placed in the specified accumulator. The previous contents of the location addressed by *E* remain unchanged.

## Store Accumulator

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



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

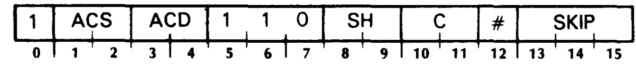


Stores the contents of a accumulator into a memory location.

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.

## Add

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



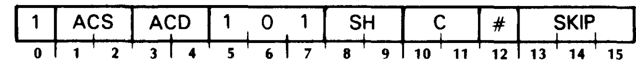
Performs unsigned integer addition and complements the carry bit if appropriate.

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

**NOTE** *If the sum of the two numbers being added is greater than $65{,}535_{10}$, the carry bit is complemented.*

## Subtract

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



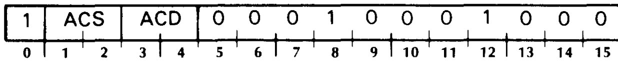
Performs unsigned integer subtraction and complements the carry bit if appropriate.

The carry bit is initialized to its specified value. The unsigned, 16-bit number in ACS is subtracted 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 result of the addition is placed in the shifter. If the operation produces a carry of 1 out of the high-order bit, the carry bit is complemented. The specified shift operation is performed and the result of the shift is placed in ACD if the no-load bit is 0. If the skip condition is true, the next sequential word is skipped.

**NOTE** *If the number in ACS is less than or equal to the number in ACD, the carry bit is complemented.*

## Decimal Add

**DAD** *acs,acd*



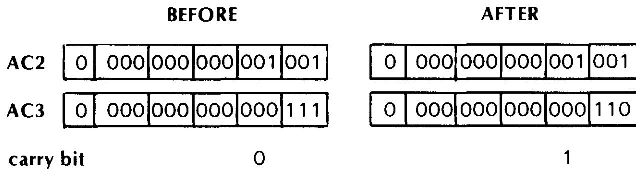

Performs decimal addition on 4-bit binary coded decimal (BCD) numbers and uses the carry bit for a decimal carry.

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

> **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.*

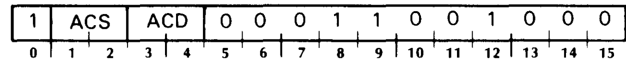
Example:

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.




## Decimal Subtract

**DSB** *acs,acd*

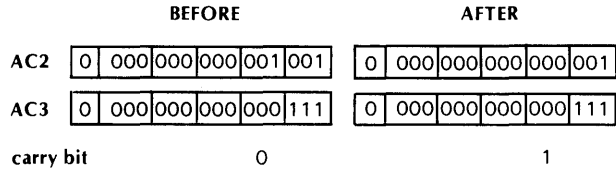



Performs decimal subtraction on 4-bit binary coded decimal (BCD) numbers and uses the carry bit for a decimal borrow.

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

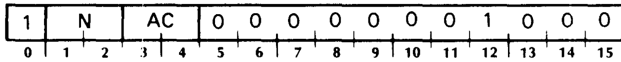
Example:

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.



## Add Immediate

**ADI** *n,ac*



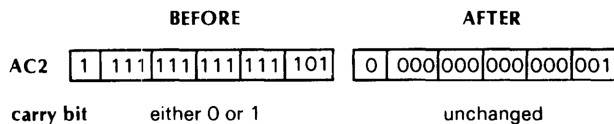

Adds an unsigned integer in the range 1-4 to the contents of an accumulator.

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

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

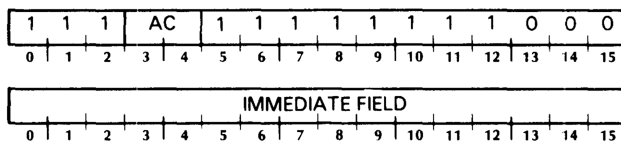
Example:

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




## Extended Add Immediate

**ADDI** *i,ac*

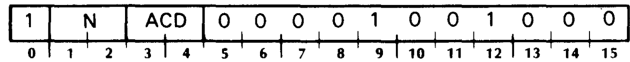



Adds a signed integer in the range $-32{,}768_{10}$ to $+32{,}767_{10}$ to the contents of an accumulator.

The contents of the immediate field are treated as a signed, 16-bit, two's complement number and added to the signed, 16-bit, two's complement number contained in the specified AC and the result is placed in the specified AC. The carry bit remains unchanged.

## Subtract Immediate

**SBI** *n,ac*



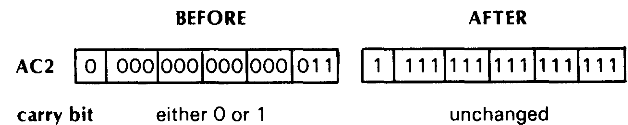

Subtracts an unsigned integer in the range 1-4 from the contents of an accumulator.

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

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

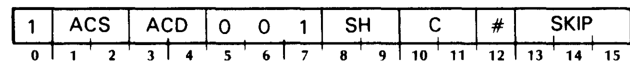
Example:

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




## Negate

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



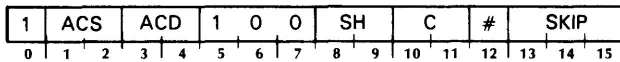

Forms the twos complement of the contents of an accumulator.

The carry bit is initialized to the specified value. The two's complement of the unsigned, 16-bit number in ACS is placed in the shifter. If the negate operation produces a carry of 1 out of the high-order bit, the carry bit is complemented. The specified shift operation is performed and the result is placed in ACD if the no-load bit is 0. If the skip condition is true, the next sequential word is skipped.

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

## Add Complement

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



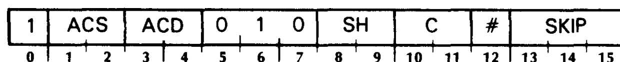

Adds an unsigned integer to the logical complement of another unsigned integer.

The carry bit is initialized to the specified value. The logical complement of the unsigned, 16-bit number in ACS is added to the unsigned, 16-bit number in ACD and the result is placed in the shifter. If the addition produces a carry of 1 out of the high-order bit, the carry bit is complemented. The specified shift operation is performed, and the result of the shift is loaded into ACD if the no-load bit is 0. If the skip condition is true, the next sequential word is skipped.

> **NOTE** *If the number in ACS is less than the number in ACD, the carry bit is complemented.*

## Move

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

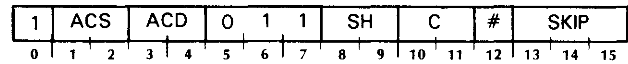



Moves the contents of an accumulator through the Arithmetic Logic Unit (ALU).

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

## Increment

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



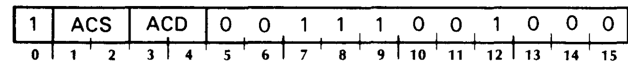

Increments the contents of an accumulator.

The carry bit is initialized to the specified value. The unsigned, 16-bit number in ACS is incremented by one and the result is placed in the shifter. If the incrementation produces a carry of 1 out of the high order bit, the carry bit is complemented. The specified shift operation is performed, and the result of the shift is loaded into ACD if the no-load bit is 0. If the skip condition is true, the next sequential word is skipped.

> **NOTE** *If the number in ACS is $177777_8$, the carry bit is complemented.*

## Exchange Accumulators

**XCH** *acs,acd*

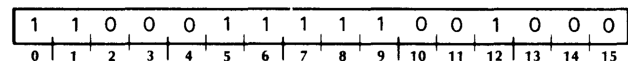



Exchanges the contents of two accumulators.

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

## Unsigned Multiply

**MUL**

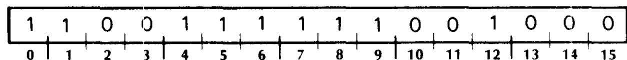



Multiplies the unsigned contents of two accumulators and adds the result to the unsigned contents of a third accumulator. The result is an unsigned 32-bit integer in two accumulators.

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

## Signed Multiply

### MULS

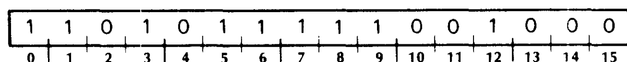



Multiplies the signed contents of two accumulators and adds the result to the signed contents of a third accumulator. The result is a signed 32-bit integer in two accumulators.

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

## Unsigned Divide

### DIV



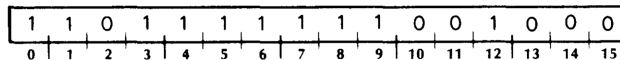

Divides the unsigned 32-bit integer in two accumulators by the unsigned contents of a third accumulator. The quotient and remainder each occupy one accumulator.

The unsigned, 32-bit number contained in AC0 and AC1 is divided 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



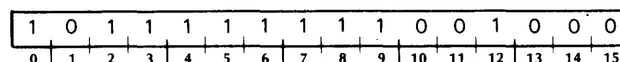

Divides the signed 32-bit integer in two accumulators by the signed contents of a third accumulator. The quotient and remainder each occupy one accumulator.

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

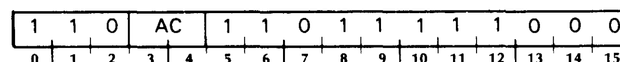



Extends the sign of one accumulator into a second accumulator and performs a *Signed Divide* on the result.

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

## Halve

### HLV *ac*




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.

## Programming Examples

The following examples illustrate the use of some of the above instructions.

### Two Accumulator-Multiple Operation Instructions

These instructions have the capability to perform multiple operations on one or two operands, often doing in one instruction what in a less sophisticated machine would take two or more. As an example, consider the problem of determining if the number in accumulator 3 is positive or negative. This could be done with two instructions as follows:

```
SUB     0,0      ;Subtract AC0 from itself, producing 0
ADCZ#   3,0,SZC  ;Skip next instruction if AC3 is
                 ; greater than or equal to AC0 (0).
--               ;Here if AC3 < 0
--               ;Here if AC3 > OR = 0
```

AC0 is subtracted from itself, producing a 0. Then the carry bit is set to 0 (Z), and the number in AC3 is complemented (ADC). The sign bit is then tested (SZC). If AC3 was positive, the carry bit will remain 0 when AC3 is complemented and the next instruction will be skipped. If AC3 was negative, the carry bit will be complemented when AC3 is complemented and no skip will occur.

The above method not only requires 2 instructions, but also destroys the contents of AC0. A better way to do this would be as follows:

```
MOVL#   3,3,SZC  ;Shift AC3 Left and test Carry bit
--               ;Here if AC3 < 0
--               ;Here if AC3 > or = 0
```

Here the *Move* instruction is used to test AC3. The contents of AC3 are shifted left one bit (L), putting the sign bit into the carry bit. The carry bit is then tested (SZC). If the carry bit is zero, the sign was zero, indicating either 0 or a positive number. Otherwise, the number is negative. The no-load option is selected (#) so that AC3 (and the carry bit) will be unchanged at the end of the operation.

### Double-Precision Arithmetic

Double-length integers are involved in both multiply and divide operations. Consequently, there is often a need to do double-precision addition and subtraction. A double-length number consists of two words concatenated into a 32-bit string. Bit 0 of the first word is the sign bit, and bits 1-31 are the magnitude in twos complement notation. Note that this means that the high-order word is in ones complement notation (i.e., negating it requires only a logical complement), unless the low-order word is all zeros. If the low-order word consists of all zeros, the high-order word is in twos complement notation.

### Double-Precision Addition

In double-precision addition, the low-order words are treated as positive integers. If a carry occurs, 1 is added to the sum of the high-order words. We add the double-precision integer in AC2 and AC3 to the double-precision integer in AC0 and AC1.

```
ADDZ    3,1,SZC  ;Set Carry to 0, add the low-order
                 ; parts, and skip if Carry = 0
INC     0,0      ;Here if Carry /= 0: add 1 to
                 ; high-order part
ADD     2,0      ;Here if Carry = 0 or after AC0 is
                 ; incremented; add high-order parts
```

In the above example, the *Add* instruction is used to add the two low- order parts, and also to test for a carry out from the low-order addition. If a carry is indicated, the *Increment* instruction adds one to one of the high-order operands. Another *Add* is used to add the high-order parts. The result is a 32-bit twos complement integer in AC0 (high-order part) and AC1 (low-order part).

### Double-Precision Subtraction

Double-precision subtraction can be performed in a manner parallel to the procedure used above for double-precision addition. A carry should occur unless the subtrahend is too large. We subtract the double-precision integer in AC2 and AC3 from that in AC0 and AC1.

```
SUBZ    3,1,SZC  ;Set Carry to 0, subtract the low-order
                 ; parts, and skip if Carry = 0
INC     0,0      ;Here if Carry /=0: add 1 to high-order
                 ; part
ADC     2,0      ;Here if Carry = 0 or after AC0 is
                 ; incremented; add complement of 1
                 ; high-order part to other
```

This method uses two instructions for some situations, and three instruction for other situations. It is possible, however, to do this executing just two instruction under all conditions. Note that incrementing the high-order part is precisely the difference between creating a ones complement and a twos complement. We can therefore get the same result this way:

```
SUBZ    3,1,SZC  ;Set Carry to 0, Subtract the low-order
                 ; parts, and skip if Carry = 0
SUB     2,0,SKP  ;Here if Carry /=0: add 2's complement
                 ; of one high-order part to the other
ADC     2,0      ;Here if Carry = 0: add 1's complement
                 ; of one high-order part to the other
```

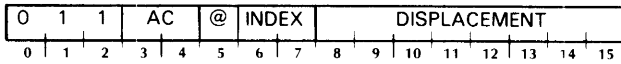
Here the first SUB instruction does the same test as was done in the previous example. If the carry does not equal zero, the twos complement of one of the high-order parts is added to the other high-order part. If the carry does equal zero, the ones complement of one of the high-order parts is added to the other high-order part.

# LOGICAL OPERATIONS

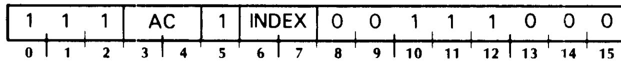
The logical instruction set performs logical operations on operands in accumulators. Three of the instructions operate on 32-bit (2 accumulator) operands, and the rest operate on 16-bit operands.

## Load Effective Address

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




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



Places an effective address in an accumulator.

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

The following applies to the *Lef* instruction ONLY:

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.

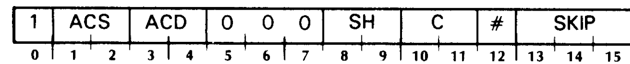
The *Lef* instruction can only be used in a mapped system, while in the user mode. With the *Lef* mode bit set to 1, all I/O and *Lef* instructions will be interpreted as *Lef* instructions. With the *Lef* MODE bit set to 0, all I/O and *Lef* instructions will be interpreted as I/O instructions.

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

**CAUTION** *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.*

## Complement

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

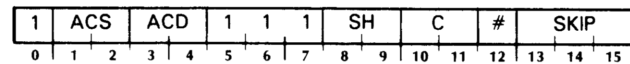



Forms the logical complement of the contents of an accumulator.

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

## AND

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

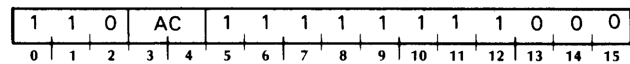



Forms the logical AND of the contents of two accumulators.

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 result bit is 0. The specified shift operation is performed and the result is placed in ACD if the no-load bit is 0. If the skip condition is true, the next sequential word is skipped.

## AND Immediate

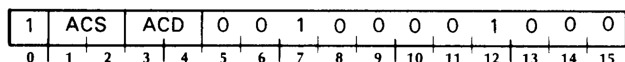
**ANDI** *i,ac*



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

## Inclusive OR

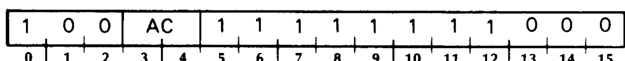
**IOR** *acs,acd*



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

## Inclusive OR Immediate

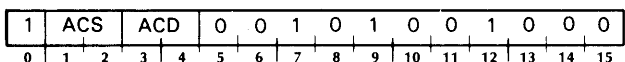
**IORI** *i,ac*



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.

## Exclusive OR

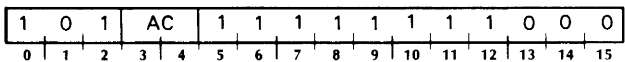
**XOR** *acs,acd*



Forms the logical exclusive OR of the contents of ACS and the contents of ACD and places the result in ACD. A bit position in the result is set to 1 if the corresponding bit positions in the two operands are unlike; otherwise, the result bit is set to 0. The contents of ACS remain unchanged.

## Exclusive OR Immediate

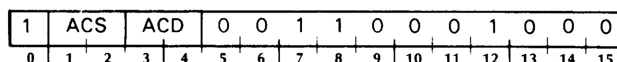
**XORI** *i,ac*



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.

## AND With Complemented Source

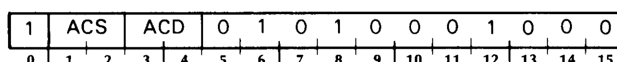
**ANC** *acs,acd*



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

## Logical Shift

**LSH** *acs,acd*



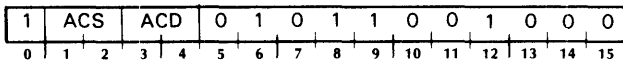
Shifts the contents of ACD either left or right depending on the number contained in bits 8-15 of ACS. The signed, 8-bit two's complement number contained in bits 8-15 of ACS determines the direction of the shift and the number of bits to be shifted. If the number in bits 8-15 of ACS is positive, shifting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

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

**NOTE** *If the magnitude of the number in bits 8-15 of ACS is greater than $15_{10}$, all bits of ACD are set to 0. The carry bit and the contents of ACS remain unchanged.*

## Double Logical Shift

**DLSH** *acs,acd*



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.

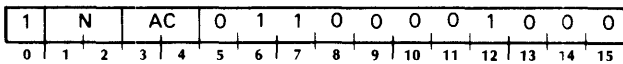
The number of bits shifted is equal to the magnitude of the number in bits 8-15 of ACS. Bits shifted out are lost, and the vacated bit positions are filled with zeroes. The carry bit and the contents of ACS remain unchanged.

**NOTES** *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.*

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

## Hex Shift Left

**HXL** *n,ac*

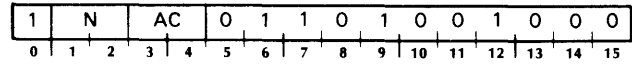


Shifts the contents of AC left a number of hex digits depending upon the immediate field *N*. The number of digits shifted is equal to N+1. Bits shifted out are lost, and the vacated bit positions are filled with zeroes. If N is equal to 3, then all 16 bits of AC are shifted out and all bits of AC are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

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

## Hex Shift Right

**HXR** *n,ac*

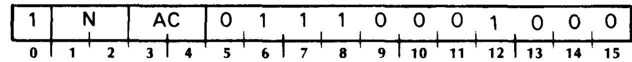


Shifts the contents of AC right a number of hex digits depending upon the immediate field, *N*. The number of digits shifted is equal to N+1. Bits shifted out are lost, and the vacated bit positions are filled with zeroes. If N is equal to 3, then all 16 bits of AC are shifted out and all bits of AC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

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

## Double Hex Shift Left

**DHXL** *n,ac*



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

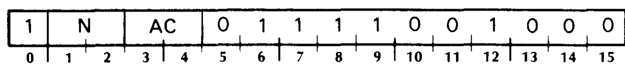
**NOTES** *If AC is specified as AC3, then AC+1 is AC0.*

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

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

## Double Hex Shift Right

**DHXR** *n,ac*




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

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

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

## Example

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

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

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

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

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

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

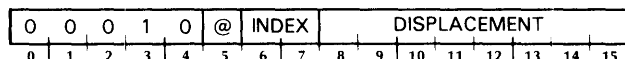
| | BEFORE | AFTER |
|---|---|---|
| AC0 | 1 111 000 111 101 111 | 1 110 000 100 000 000 |
| AC1 | 0 001 000 011 101 111 | 0 001 000 011 101 111 |

# CONDITIONAL INSTRUCTION SET

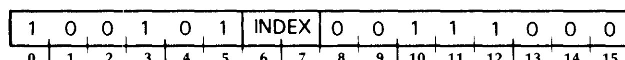
The following instructions test a specified value and then skip the next sequential instruction if the test result is true. If the test result is not true, some instructions will merely permit the next sequential instruction to be executed, and some will do some other task, but in either case, the skip does not take place.

## Increment And Skip If Zero

**ISZ** *[@]displacement[,index]*



**EISZ** *[@]displacement[,index]*

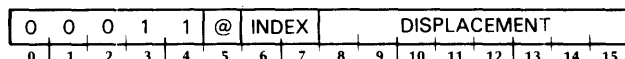


Increments the addressed word, then skips if the incremented value is zero.

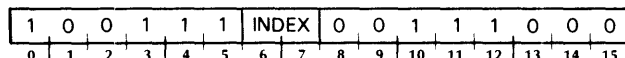
The word addressed by *E* is incremented by one and the result is written back into that location. If the updated value of the location is zero, the next sequential word is skipped.

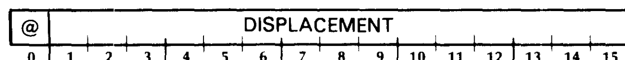
## Decrement And Skip If Zero

**DSZ** *[@]displacement[,index]*



**EDSZ** *[@]displacement[,index]*



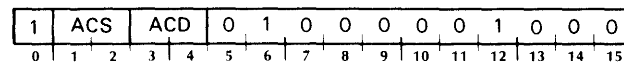


Decrements the addressed word, then skips if the decremented value is zero.

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

## Skip If ACS Greater Than ACD

**SGT** *acs,acd*

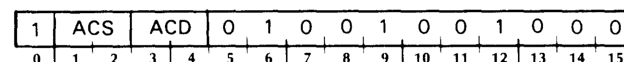


Compares two signed integers in two accumulators and skips if the first is greater than the second.

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 If ACS Greater Than Or Equal to ACD

**SGE** *acs,acd*



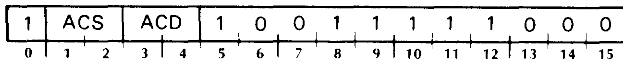
Compares two signed integers in two accumulators and skips if the first is greater than or equal to the second.

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

> **NOTE** *The SKIP IF ACS GREATER THAN ACD and SKIP IF ACS GREATER THAN OR EQUAL TO ACD instructions treat the contents of the specified accumulators as signed, two's complement integers. For comparison of unsigned integers, the SUBTRACT and ADD COMPLEMENT instructions may be used. Use of these instructions for comparison is described in Appendix H.*

## Compare To Limits

**CLM** *acs,acd*




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.

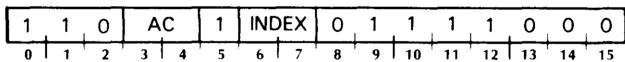
Compares the signed, two's complement integer in ACS to two signed, two's complement limit values, L and H. If the number in ACS is greater than or equal to L and less than or equal to H, the next sequential word is skipped. If the number in ACS is less than L or greater than H, the next sequential word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

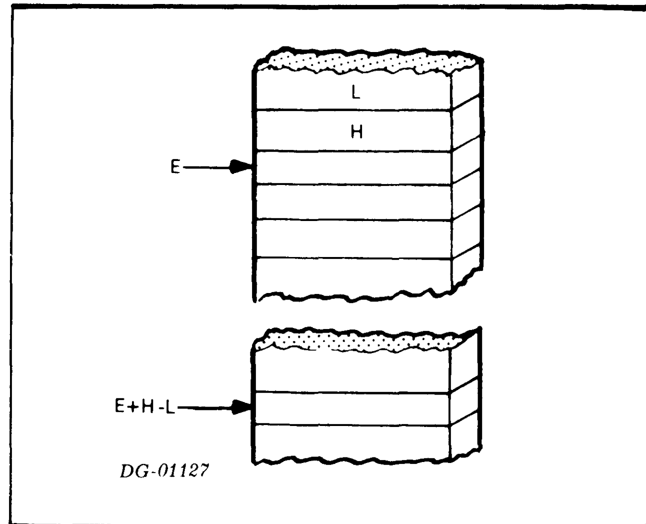
## Dispatch

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



Conditionally transfers control to an address selected from a table.

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



DG-01127

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

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

## Programming Examples

Consider an interactive program which accepts a number of one-letter commands from the user's terminal. We assume that the following routine is called with one ASCII character already in bits 8-15 of AC0.

CMDTB is the address of a 26-word table in main memory, which is preceded by limit words containing the codes for A and Z. ILLCH is the address of a routine to be executed if the user types some character other than a letter. BADCM is the address of a routine to be executed if the user types a letter which has no command function. Command letters may be scattered randomly through the alphabet.

```
        ...
        ...                  ;Character now in AC0
        DSPA    0,CMDTB  ;Go to proper routine
        JMP     ILLCH    ;Here if the user didn't type a letter
        ...

        ;This is the dispatch table

        101              ;Lower limit - octal for "A"
        132              ;Upper limit - octal for "Z"
CMDTB:  ACOMD            ;Address of routine for "A" command
        BCOMD            ;Address for "B" command
        BADCM            ;"C" is not a legal command
        BADCM            ;Neither is "D"
        ECOMD            ;But "E" is
        ...
        ...
        ZCOMD            ;"Z" - the last command
```

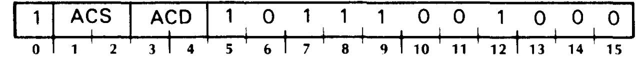
Note that only a single instruction was needed to analyze the character. Without the *Dispatch* instruction, testing each letter for validity might require a large number of comparison operations. The *Dispatch* instruction requires some additional memory for the dispatch table, but saves considerable time.

# BYTE MANIPULATION

The instructions that manipulate bytes are often used to perform character operations. Use of the SWAP option of the two accumulator- multiple operation instructions gives access to both bytes of the word.

## Load Byte

**LDB** *acs,acd*



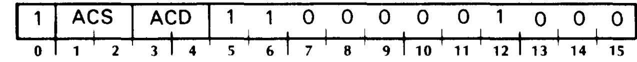
| 1 | ACS | | ACD | | 1 | 0 | 1 | 1 | 1 | 0 | 0 | 1 | 0 | 0 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |

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

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

## Store Byte

**STB** *acs,acd*



| 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.

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

## Programming Example

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

```
        LDA     2,CON6   ;Get count
        STA     2,CNT    ;Store it
        LDA     2,MASK   ;Get mask for char
                         ; and shift count
LOOP:   DLSH    2,0      ;Shift one octal
                         ; digit
        HXR     1,1      ;Shift AC1 4 bits
                         ; right
        MOVZR   1,1      ;Shift AC1 1 bit
                         ; right
        IOR     2,1      ;OR in bits for
                         ; character
        MOVS    1,1      ;Put char in low-
                         ; order byte
        STB     3,1      ;Store byte
        DSZ     CNT      ;Done?
        JMP     .+2      ;No
        JMP     OUT      ;Yes
        SBI     1,3      ;Decrement AC3 by 1
        JMP     LOOP     ;Do next digit
CON6:   6
CNT:    0
MASK:   030375           ;Char mask in hi
                         ; byte. Shift count
                         ; in low byte
OUT:    ...
        ...
        ...
```

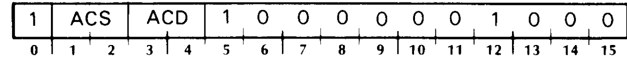
When the routine is finished, and control is transferred to the location *OUT*, the integer has been converted to octal and the byte pointer in AC3 points to the high-order byte of the result.

# BIT MANIPULATION

The instructions in this set operate on individual bits. Bits can be set to 0 or 1, and lead bits can be located. In addition, bits can be tested, causing a skip of the next word if specified conditions are true.

## Set Bit To One

**BTO** *acs,acd*

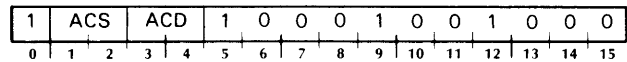



The two accumulators form a bit pointer. Sets the indicated bit to 1.

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

## Set Bit To Zero

**BTZ** *acs,acd*

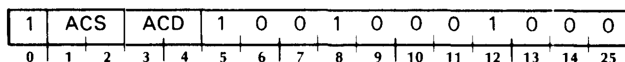



The two accumulators form a bit pointer. Sets the addressed bit to 0.

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

## Skip On Zero Bit

**SZB** *acs,acd*

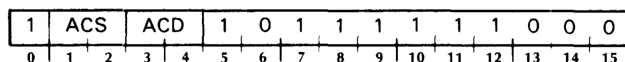


The two accumulators form a bit pointer. If the addressed bit is zero, the next sequential instruction is skipped.

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

## Skip On Non-Zero Bit

**SNB** *acs,acd*

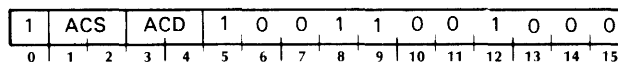


The two accumulators form a bit pointer. If the addressed bit is 1, the next sequential instruction is skipped.

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

## Skip On Zero Bit And Set To One

**SZBO** *acs,acd*



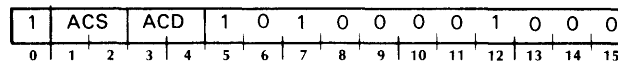
The two accumulators form a bit pointer. Sets the addressed bit to 1. If it was 0 originally, the next sequential word is skipped.

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

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

## Locate Lead Bit

**LOB** *acs,acd*



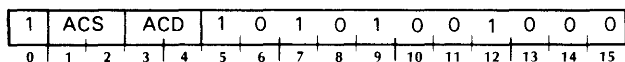
Adds a number equal to the number of high-order zeros in the contents of the first accumulator to the contents of the second accumulator.

The contents of ACS are inspected for high-order zeroes. A number equal to the number of high-order zeroes is added 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*



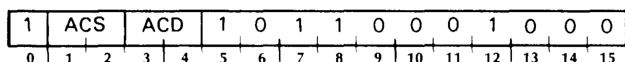

Performs a LOCATE LEAD BIT instruction, and sets the lead bit to 0.

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

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

## Count Bits

**COB** *acs,acd*




Adds a number equal to the number of ones in one accumulator to the contents of the second accumulator.

The contents of ACS are inspected for 1's. A number equal to the number of 1's in ACS is added 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.*

## Programming Example

The routine below uses the *Skip on Non-Zero Bit* instruction to detect a *delimiter* in a string of input characters. Delimiters might include carriage return, line feed, null, rubout, or any arbitrary set of characters. The routine will stop after reading a delimiting character. Without bit manipulation, testing a string for delimiters would be a complex operation.

In this example we use a table **DLMTB**, consisting of 16 words containing one bit for each possible character value. A character is declared to be a delimiter by setting its bit in the table to 1. For instance, since carriage return has a value of $15_8$, $(13_{10})$, bit 13 in the first word of **DLMTB** must be 1 to declare it a delimiter.

**GTCHR** is a subroutine which places the next character from the input stream into AC1. AC2 is used for a byte pointer to the area in memory in which to store the characters as they are processed.

AC0 is loaded with the address of **DLMTB** to serve as the first word of the bit pointer. AC1 contains the second word of the bit pointer. The 4 high-order bits of the character in AC1 select one of the 16 words of **DLMTB**, and the 4 low-order bits select the bit to be tested.

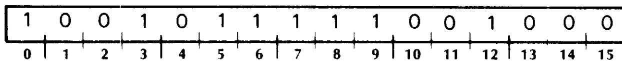
```
        ...
READ:   LEF     0,DLMTB ;Load AC0 with address of table
        LDA     2,BTPTR ;Load AC2 with destination byte pointer

LOOP:   JSR     GTCHR   ;Get next character in AC1
        STB     2,1     ;Store it in destination area
        INC     2,2     ;Increment byte pointer
        SNB     0,1     ;Check bit: a delimiter?
        JMP     LOOP    ;If not, Get next character
        ...             ;Here if delimiter found (Exit)
```

# BLOCK MANIPULATION

Two instructions are provided in the S/130 for the rapid movement of a block of data from one area of memory to another. These instructions move from 1 to 32,768 words in one operation. The *Block Add and Move* instruction also adds a constant to each word as it is moved. This allows easy relocation of address constants.

## Block Add And Move

### BAM



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

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

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

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

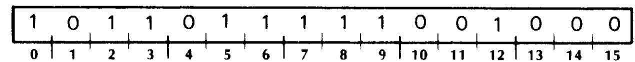
Words are moved in consecutive, ascending order according to their addresses. The next address after $77777_8$ is 0 for both fields. The fields may overlap in any way.

**NOTE** *Because of the potentially long time that may be required to perform this instruction in relation to I/O requests, this instruction is interruptable. If a* Block Add And Move *instruction is interrupted, the program counter is decremented by one before it is placed in location 0 so that it points to the* Block Add And Move *instruction. Because the addresses and the word count are updated after every word is stored, any interrupt service routine that returns control to the interrupted program via the address stored in memory location 0 will correctly restart the* Block Add And Move *instruction.*

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

## Block Move

### BLM



Moves memory words from one location to another.

The BLOCK MOVE instruction is equivalent to the BLOCK ADD AND MOVE instruction in all respects except that no addition is performed and AC0 is not used.

**NOTE** *The BLOCK MOVE instruction is interruptable in the same manner as the BLOCK ADD AND MOVE instruction.*

## Programming Example

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

```
        LDA     2,TBLAD   ;Put address of
                          ; table in AC2
        INC     2,3       ;Put address of
                          ; table +1 in AC3
        SUBO    0,0       ;SET AC0 to 0
        STA     0,0,2     ;Set first word
                          ; of table to 0
        INC     0,0       ;AC0=addend of 1
        MOV     0,1       ;Put 16 (decimal)
        HXL     1,1       ;In AC1
        BAM               ;Create table
        JMP     TABLE+21  ;Jump around table
                          ; 17 (decimal)=
                          ; 21 (octal)
TBLAD:  TABLE             ;Address of table
TABLE:  .BLK 21           ;Reserve 21 (octal)
                          ; words for table
        ...
        ...
```

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

# THE STACK

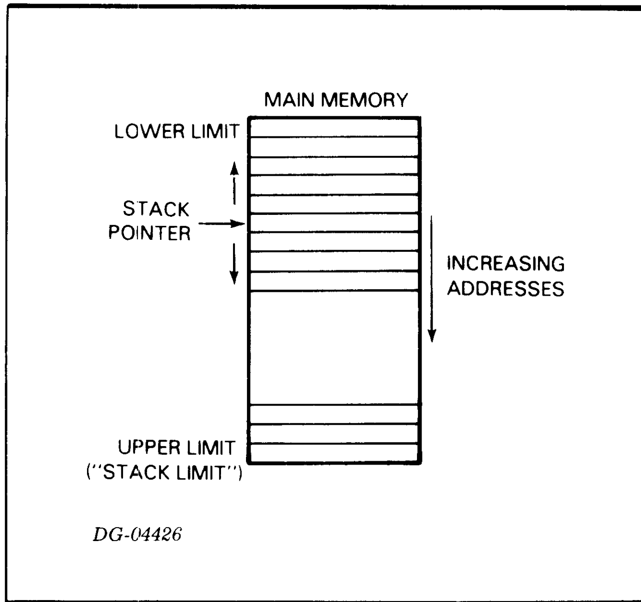
## Introduction

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 S/130 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. The *Vector* instruction can put onto the stack a combination of a return block and individual words totaling up to 10 words, and the floating point instruction set uses a return block of 18 words.

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, and 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.

The value of the stack pointer when the stack is set up 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 is indicated. The stack limit is contained in location $42_8$.

### 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 that 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 $77777_8$.

**NOTE** *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 *instruction - see details in the discussion of the* Save *instruction below). 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 is indicated 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 then not occur unless the program underflows the stack and 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

After every operation that pushes data onto the stack, a check is made for overflow. 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.

Determine the nature of the fault by looking at bits 1-15 of the stack pointer and the stack limit. There are three possibilities:

- If the address contained in the stack pointer is not greater than the address in the stack limit, then the error was a stack overflow error resulting from the execution of a *Save* instruction.
- If the address in the stack pointer is greater than the address in the stack limit by a value greater than 5, then the error was a stack overflow error.
- If the address in the stack pointer is greater than the address in the stack limit by exactly 5, then the error was a stack underflow error.

## 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

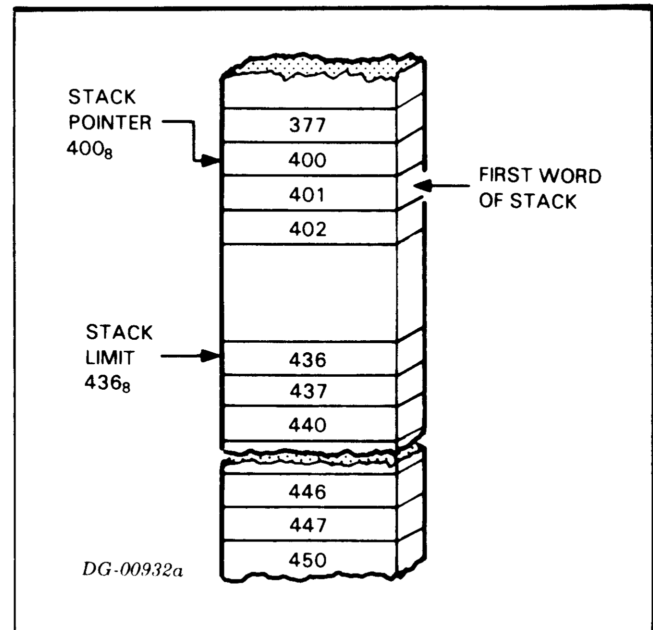
Initialize the stack fault address to the address of the routine that is to receive control in the event of a stack overflow or underflow. Bit 0 may be set to 1 to indicate an indirect address.

### Frame Pointer

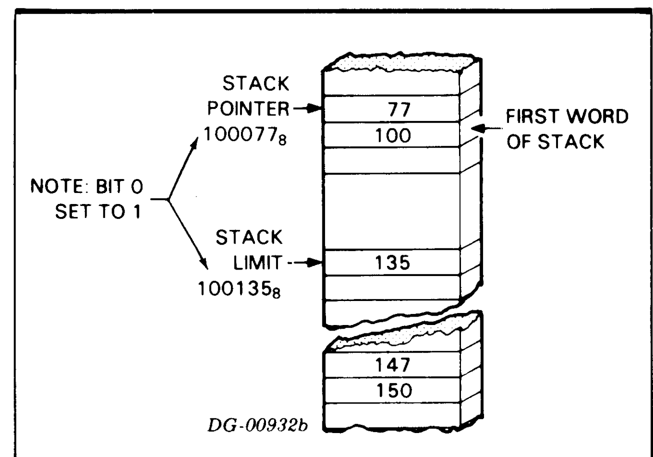
If the main user program is going to use the frame pointer, initialize it to the same value as the stack pointer. Otherwise, the frame pointer can be initialized in a subroutine by the *Save* instruction.

## Examples

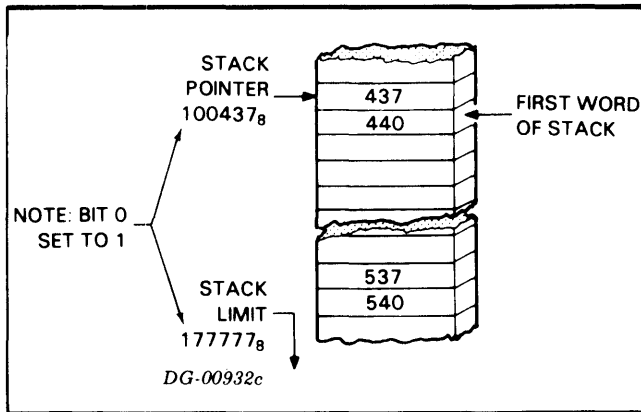
Stack area of $50_8$ words with overflow and underflow protection




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



Stack area of $100_8$ words with no protection




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

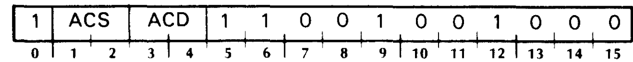
```
        .TITL   STACK

        .EXTN   STKHR  ;Declare STKHR 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
        @STKFT         ;Stack fault address

STKFT:  STKHR          ;Address of stack handler

        .END
```

## STACK INSTRUCTIONS

The instructions for use of the stack are listed below.

### 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 |

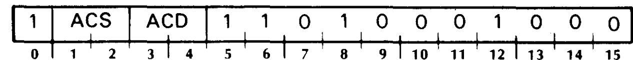
Pushes the contents of 1 to 4 accumulators onto the stack.

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 is completed.

### 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 |

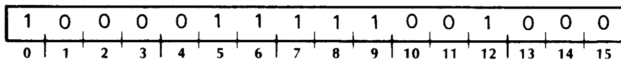
Pops 1 to 4 words off the stack and places them in the indicated accumulators.

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 completed.

## Push Return Address

### PSHR



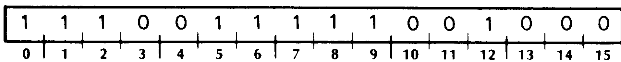

Pushes the address of the instruction after the next sequential instruction onto the stack.

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

> **NOTE** *The* Push Return Address *instruction adds 2 to the program counter (rather than 1) to allow room for a* Jump *or* Jump to Subroutine *instruction.*

## Save

### SAVE *i*



Saves the information required by the RETURN instruction.

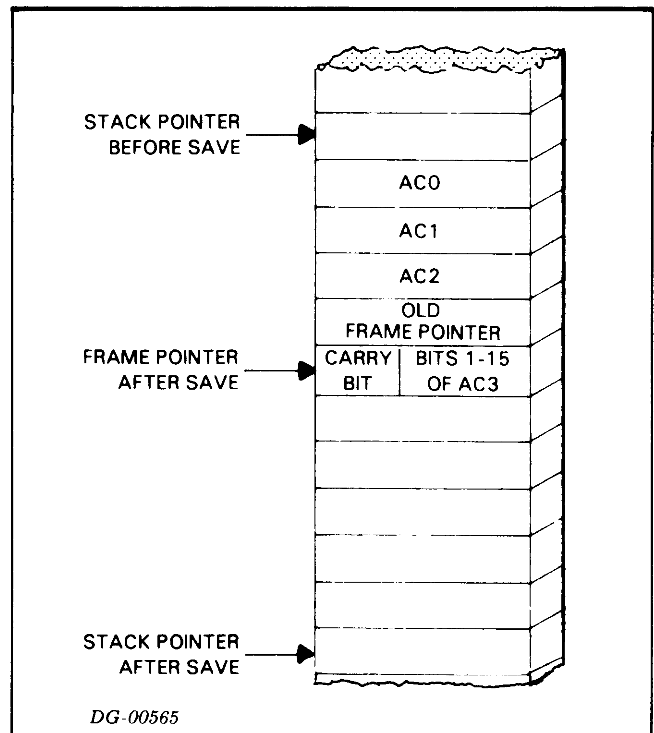
A return block is pushed onto the stack. After the fifth word of the return block is pushed, the value of the stack pointer is placed in the frame pointer and in AC3. The 16-bit unsigned integer (called the *frame size)* contained in the immediate field is added to the stack pointer. The format of the five words pushed is as follows:

| Word No. Pushed | Contents |
|---|---|
| 1 | AC0 |
| 2 | AC1 |
| 3 | AC2 |
| 4 | Frame pointer before the SAVE. |
| 5 | Bit 0 = carry bit<br>Bits 1-15 =<br>bits 1-15 of AC3 |

**NOTES** *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 the* Save *instruction is executed, a check for stack overflow is performed. If execution of the instruction would result in a stack overflow, the instruction is not executed, and control is transferred 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.*



## Programming Example

The following example transfers control to the subroutine **LOOP**, saves the return information, and allocates a 6-word block in the stack for use by **LOOP**. The variable **NUM2** is placed in the first of these 6 words. Note that the value of **NUM1**, left in AC0 by the calling procedure, is retrieved by the called procedure, which reaches into the return block, using the frame pointer as a reference.

```
        LDA   0,NUM1 ;Put NUM1 in AC0
        JSR   LOOP   ;Go to loop subroutine
LOOP:   SAVE  6      ;Save return info, allocate 6 words
                     ; for temporary storage
        LDA   2,FP   ;Get frame pointer
        LDA   1,-4,2 ;Get contents of stack word containing
                     ; previous contents of AC0, using
                     ; frame pointer as reference.
        LDA   3,NUM2 ;Get other number, put in AC3
        ADD   1,3    ;Add NUM1 + NUM2
        STA   3,1,2  ;Put sum in first word of temporary
                     ; storage area.
```

## Modify Stack Pointer

**MSP** *ac*

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

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

The signed twos-complement number in AC is added to the stack pointer. If the result is less than the stack limit, the result is placed in the stack pointer.

If the result is greater than the stack limit, control is transferred 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.

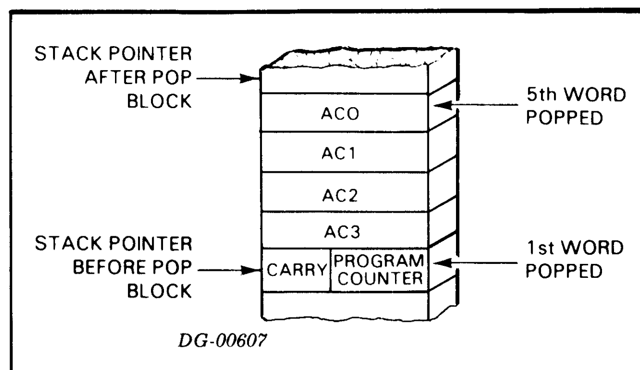
## Pop Block

**POPB**

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

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:



DG-00607

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

> **NOTE** *If the I/O interrupt 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.*

## Return

**RTN**

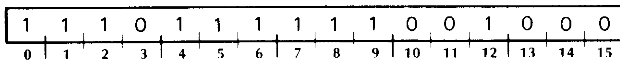
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 0 | 1 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 1 | 0 | 0 | 0 |

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 pointer.

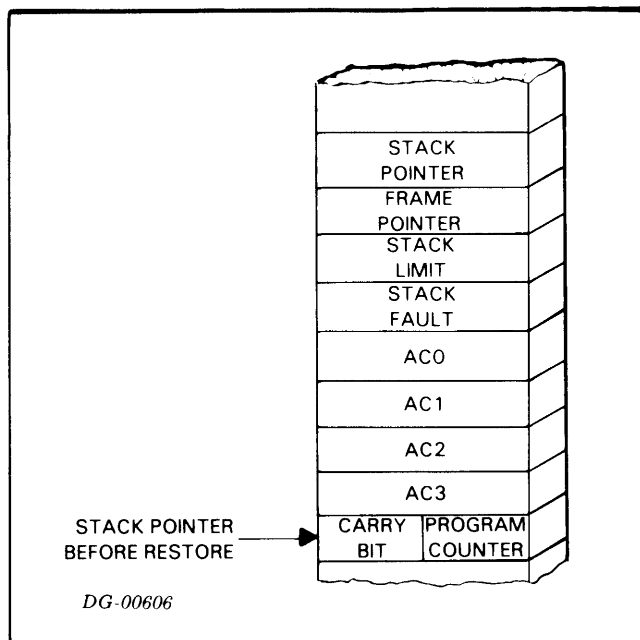
## Restore

### RSTR




Returns control from certain types of I/O interrupts.

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




DG-00606

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

**NOTES** *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.*

*No check for stack underflow is performed as part of the* Restore *instruction.*

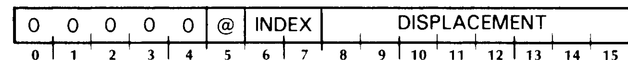
*See the* Vector *instruction section (Chapter V) for an example using this instruction.*

# PROGRAM FLOW ALTERATION

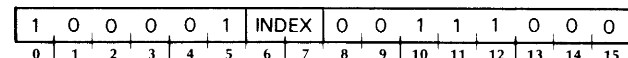
The processor will retrieve and execute instructions from sequentially addressed memory locations unless directed to do otherwise. The following instructions alter this sequential flow by placing a new value in the program counter. Sequential operation will then continue with the instruction addressed by this new value.

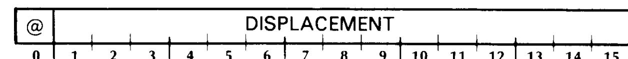
## Jump

**JMP** *[@]displacement[,index]*




**EJMP** *[@]displacement[,index]*

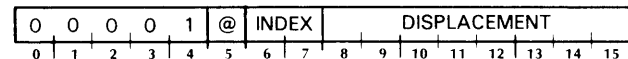


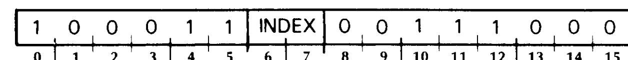

Computes the effective address, *E* and places it in the program counter. Sequential operation continues with the word addressed by the updated value of the program counter.

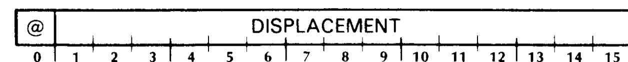
## Jump To Subroutine

**JSR** *[@]displacement[,index]*




**EJSR** *[@]displacement[,index]*



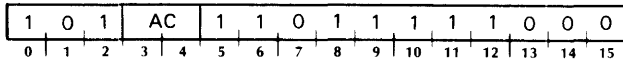

Increments and stores the value of the program counter in AC3, and then places a new address in the program counter.

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

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

## Execute

### XCT *ac*



Executes the instruction contained in AC as if it were in main memory in the location occupied by the EXECUTE instruction. If the instruction in AC is an EXECUTE instruction which executes the instruction 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.

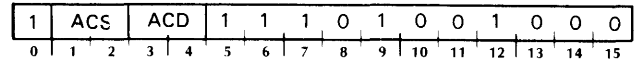
Because of the possibility of AC containing an EXECUTE instruction, this instruction is interruptable. An I/O interrupt can occur immediately prior to each time the instruction in AC is executed. If an I/O interrupt does occur, the program counter in the return block pushed on the system stack points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

**NOTE** *The results of XCT are undefined if the specified accumulator contains an instruction that modifies that same accumulator. For example:*

```
        LDA     0,FOO
        XCT     0           ;UNDEFINED
        JMP     ON

FOO:    FFAS    0,0
```

## System Call

### SYC *acs,acd*



Pushes a return block and indirectly places the address of the SYSTEM CALL handler in the program counter.

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

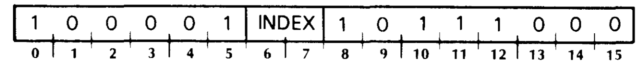
**NOTES** *If both accumulators are specified as AC0, no return block is pushed on the stack. The contents of AC0 remain unchanged. If either of the accumulators specified is not AC0, then no special action is taken. The contents of the specified accumulators remain unchanged.*

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

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

## Push Jump

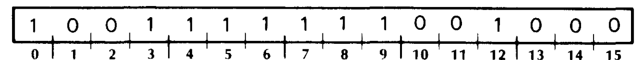
### PSHJ *[@]displacement[,index]*



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.

## Pop PC And Jump

### POPJ



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

# SUBROUTINE CALLS AND RETURNS

## Introduction

A subroutine is a section of program code which is called by some other program, and which, upon completion, transfers control back to the program calling it. Almost all well-structured programs of any size are broken down into subroutines to improve their efficiency.

The S/130 provides a number of instructions which support subroutine *linkage*. In this section, we discuss some general properties of subroutine calls and returns, and then consider some examples.

## Properties of Subroutine Calls and Routines

Calling and called routines communicate by means of *arguments* whose values or addresses are passed to the subroutine by the calling program. The arguments may be numbers which are used by the subroutine directly, or they may be addresses where the desired values are found. They may also be addresses where the subroutine will deposit results.

The subroutine may also use some main memory for *local* variables; this memory is needed while the subroutine is running, but may be used for other purposes at other times. Local storage is often used to save the original contents of accumulators before they are used by the subroutine. The subroutine can then restore the contents of these accumulators before returning to the calling program.

Subroutines can also call other subroutines *(nesting)*, or a subroutine can call itself *(recursion)*. A subroutine which calls itself is called a *recursive* subroutine.

## Programming Examples

### Example 1

The simplest method of calling a subroutine is by use of the **JSR** or **EJSR** instruction. This instruction places the contents of the program counter into AC3 and jumps to the specified address. AC3 will contain the address of the instruction following the **JSR** or **EJSR**, which is the return address. The subroutine returns by executing a **JMP 0,3**.

You can pass arguments when using this procedure by placing the values to be passed in the words following the **JSR** or **EJSR**. The subroutine can then use AC3 as an index register to reference those words, which are called *in-line* arguments because they appear in-line with the program code, rather than in a separate data area.

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

```
        ...                 ;Main program code
        JSR     FUNC        ;Call subroutine
        NUMBR               ;Number to use
        ANSWR               ;Address to place result
        ...                 ;Rest of main program

ANSWR:  ...                 ;Place to put result

FUNC:   LDA     0,0,3       ;Load argument into ACO
        ...                 ;Process it
        ...                 ;(Result placed in AC2)
        STA     2,@1,3      ;Store it back in calling program
        ADI     2,3         ;Adjust return address
        JMP     0,3         ;Return
```

The subroutine linkage shown above is simple and fast. It does not support nesting or recursion, nor does the subroutine use any local memory. Also, the action of **JSR** requires that AC3 be reserved for the return address.

### Example 2

The S/130 stack facility can improve the efficiency of subroutine linkage in several ways. If the accumulators do not need to be preserved, arguments can be passed via the accumulators, and you can use **PSHJ** to call subroutines and **POPJ** to return. Nesting and recursion are possible with **PSHJ/POPJ** since the last-in/first-out operation of the stack insures that return addresses will be preserved.

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

```
        ...                ;Input argument (N) in AC0
        PSHJ    FACT       ;Call subroutine
        ...                ;Output result in AC1
        ...                ; (Result = 0 if overflow)

FACT:   MOVZR#  0,0,SNR    ;Is input argument = 0 or 1?
        JMP     LTTLE      ;Yes, output is 1
        PSH     0,0        ;No, output = N*(N-1)! Store N in stack
        SBI     1,0        ;Decrement argument by 1
        PSHJ    FACT       ;Loop around to put proper stuff
                           ; in stack
        POP     2,2        ;Get argument from stack
        SUB     0,0        ;Need to clear AC0 for integer multiply
        MUL                ;Multiply N*(N-1)!
        MOV#    0,0,SZR    ;Overflow? (If AC0 not 0)
        SUB     1,1        ;Yes, signal overflow with 0 result
        POPJ               ;Return to next address in stack

LTTLE:  SUBZL   1,1        ;Put 1 in AC1
        POPJ               ;Return to addr in stack
```

### Example 3

The use of the *Save* and *Return* instruction with the **JSR** provides a powerful subroutine linkage that supports nesting and recursion with local variables stored on the stack, and automatic saving of the accumulators.

Below is a subroutine which is to be called by **JSR**. It uses *Save* to save the accumulators on the stack and reserve five words of the stack for local variables. It then loads the words pointed to by two in-line arguments into the first two words of the local memory. The subroutine terminates by executing a **RTN**, which restores the original condition of the accumulators and stack.

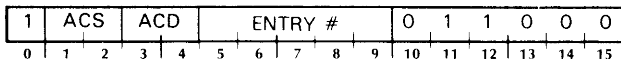
```
SUBRT:  SAVE    5          ;Save AC'S, reserve 5 words for
                           ; local storage
                           ; (AC3 now points to last word pushed
                           ; onto stack)
        LDA     2,0,3      ;Load address of arguments into AC2
        LDA     1,@0,2   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

## EXTENDED OPERATION FEATURE

The extended operation feature (XOP) provides an efficient method of transferring control to and from procedures. When used with the WCS, the XOP feature enables the user to transfer control to any one of 16 entry points in WCS. This permits the user to implement and execute his own specialized instructions.

### Extended Operation

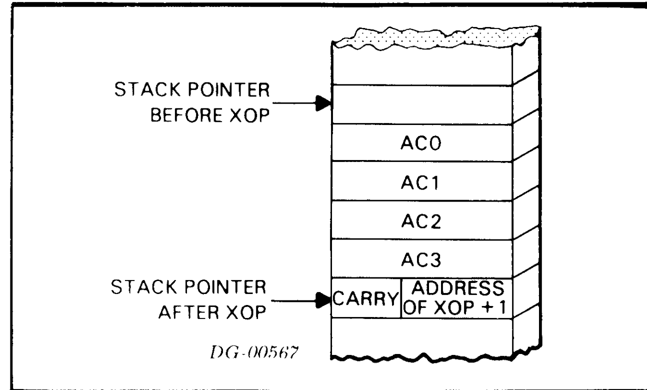
**XOP** *acs, acd, entry number*




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

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

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




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

When WCS is not installed, the XOP1 instruction, described in the next section under Writeable Control Store, serves the same function as the EXTENDED OPERATION instruction above with the exception that $32_{10}$ is added to the entry number before it is added to the XOP origin.

# CHAPTER IV
# OPTIONAL FEATURES OF THE ECLIPSE S/130

## INTRODUCTION

In this chapter we discuss the Memory Allocation and Protection unit (MAP), the floating point and character instruction sets, and the Writable Control Store (WCS).

## MEMORY ALLOCATION AND PROTECTION

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

The S/130 MAP unit provides the hardware necessary to control and use more than 32K of physical memory. A MAP is useful for systems with two users and required for systems with more than two users. In addition, the MAP provides protection functions which help protect the integrity of a large system.

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

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

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

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

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

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

**Page** - $1024_{10}$ ($2000_8$ ) words in memory.

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

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

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

## Address Translation

The primary function of the MAP is address translation. In the S/130, 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 S/130 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 S/130. *User maps* translate logical addresses to physical addresses when memory reference instructions are encountered in the user's program. *Data channel maps* translate logical addresses to physical addresses when data channel devices address the memory.

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

Separate data channel maps are needed because data channel devices can access memory without direct control from the user's program. There is thus no assurance that the proper user map would still be enabled at the time of the data channel request. The MAP can hold four data channel maps. Enabling data channel mapping enables all four data channel maps at the same time. The choice of which map is used for data channel references is made by the I/O controller making the reference. Those controllers not equipped to make this distinction use data channel map A by default. See the *Programmer's Reference Manual - Peripherals (DGC No. 015-000021).*

## Supervisor Mode

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

## MAP Protection Capabilities

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

The various types of protection available in the S/130 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 pages of logical addresses required by the user's program are linked to pages of physical addresses. The remaining (unused) logical pages 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 pages 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 page of addresses is write protected when the map is specified. Write protection can be enabled or disabled at any time by the supervisor.

### Indirect Protection

An indirection loop occurs when the effective address calculation follows a chain of indirect addresses and never finds a word with bit 0 set to 0. Without indirect protection, the CPU would be unable to proceed with any further instructions, thus effectively halting the system.

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

### I/O Protection

I/O protection protects the I/O devices in the system from unauthorized access. In many systems, all I/O operations are performed through operating system calls. Clearly, it is undesirable to permit individual users to execute I/O instructions, since this will interfere with the operating system. If a user with I/O protection enabled attempts to execute an I/O instruction, a protection fault will occur. I/O protection can be enabled or disabled at any time.

## MAP Protection Faults

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

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

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

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

## Load Effective Address Mode

The *Load Effective Address* instruction uses the same instruction codes as some of the I/O instructions. Without some other indication, the S/130 would have no way of knowing which instruction was intended. The MAP therefore has a *Lef* mode bit, which switches the mode of the S/130 from *Lef* mode to I/O mode. When the *Lef* mode bit is 1, all I/O format instructions are interpreted as *Load Effective Address* instructions. When the *Lef* mode bit is 0, all I/O format instructions are interpreted as I/O instructions.

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

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

## Initial Conditions

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

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

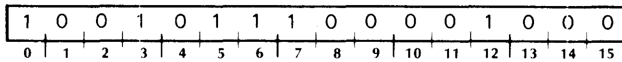
## MAP INSTRUCTIONS

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

> **CAUTION** *MAP instructions* can *be executed in the user mode if I/O protection and* Lef *mode are disabled for that user. When executed in the user 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.*

### Load Map

#### LMP



Loads successive words from memory into the MAP where they are used to define a user or data channel map.

The number of words to be loaded and the address of the beginning of the fields are contained in accumulators 1 and 2. Which address translation function is being loaded is determined by the contents of the map filed in the MAP status register.

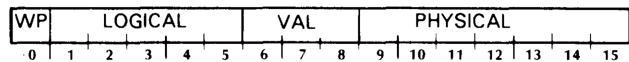
AC0 must contain 0. AC1 must contain an unsigned integer which is the number of words to be loaded into the MAP. Bits 1-15 of AC2 must contain the address of the first word to be loaded. If bit 0 of AC2 is 1, the instruction follows the indirection chain and places the resultant effective address in AC2. AC3 is ignored and its contents remain unchanged.

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

This instruction is interruptable in the same manner as the BLOCK ADD AND MOVE instruction. If this instruction is issued while in the user mode, 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. Which map is to be affected by a LOAD MAP instruction is determined by the contents of the MAP field in the MAP status register. This field can be altered by both the LOAD MAP STATUS and INITIATE PAGE CHECK instructions.

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

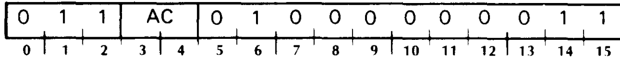


| Bits | Name | Contents |
|---|---|---|
| 0 | Write Protect | If 1, this logical page will be write protected when write protection is enabled. |
| 1-5 | Logical Page | This is the number of the logical page which is to be mapped. |
| 6-8 | Validity | These bits must be set to 1 only when declaring a logical block invalid. Otherwise, they must be set to 0. |
| 9-15 | Physical Page | This is the number of the physical page of memory that will hold the logical page defined by bits 1-5. |

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

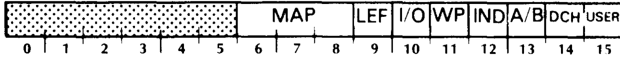
## Load Map Status

**DOA** *ac,MAP*



Defines the parameters of a new map.

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

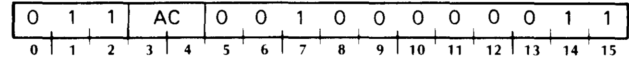


| Bits | Name | Contents |
|---|---|---|
| 0-5 | --- | Reserved for future use. |
| 6-8 | Map | Specify which map will be loaded by the next LOAD MAP instruction as follows:<br>000 User A<br>001 Reserved for future use<br>010 User B<br>011 Reserved for future use<br>100 Data channel A<br>101 Data channel C<br>110 Data channel B<br>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.<br>If 1, the next user map enabled will be that for user B. |
| 14 | DCH Enable | If 1, the mapping of data channel addresses will be enabled immediately after this instruction. |
| 15 | User Enable | If 1, mapping of CPU addresses will commence with the first memory reference after the next indirect reference or return type instruction. |

**NOTE** *If the* Load Map Status *instruction sets the User Enable bit to 1, this inhibits the interrupt system and the MAP waits for either an indirect reference or a 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 *or* Restore *instruction that does not cause a stack fault.*

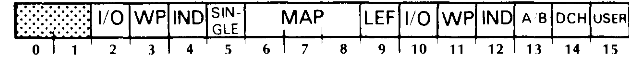
## Read Map Status

**DIA** *ac,MAP*



Reads the status of the current map.

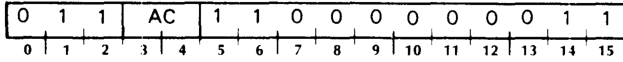
The contents of the MAP status register are placed in the specified AC. The previous contents of the specified AC are lost. The format of the information placed in the specified AC is as follows:



| Bits | Name | Contents |
|---|---|---|
| 0-1 | --- | Reserved for future use. |
| 2 | I/O | If 1, the last protection fault was an I/O protection fault. |
| 3 | WP | If 1, the last protection fault was a write protection fault. |
| 4 | IND | If 1, the last protection fault was an indirect protection fault. |
| 5 | Single Cycle | If 1, the last map reference was a MAP SINGLE CYCLE instruction. |
| 6-8 | Map | Specify which map will be loaded by the next LOAD MAP instruction as follows:<br>000 User A<br>001 Reserved for future use<br>010 User B<br>011 Reserved for future use<br>100 Data channel A<br>101 Data channel C<br>110 Data channel B<br>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 the user map for user A.<br>If 1, the last LOADMAP STATUS instruction enabled the user map for user 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. |

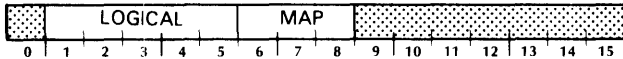
## Initiate Page Check

**DOC** *ac,MAP*



Identifies a logical page. The *Page Check* instruction will find the corresponding physical page.

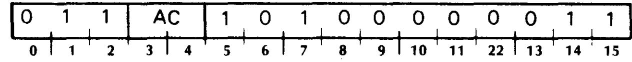
Transfers the contents of the specified AC to the MAP for later use by the *Page Check* or *Load Map* instruction. Leaves the contents of the specified AC unchanged. The format of the specified AC is as follows:



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

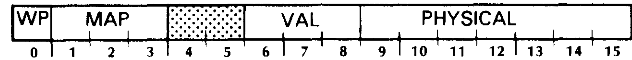
## Page Check

**DIC** *ac,MAP*



Provides the identity and some characteristics of the physical page corresponding to the logical page identified by the immediately preceding *Initiate Page Check* instruction.

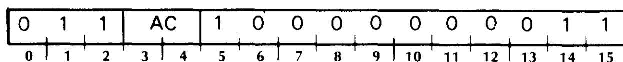
Places the number of the physical page which corresponds to the logical page specified by the preceding *Initiate Page Check* or *Load Map Status* instruction in bits 9-15 of the specified AC. Places additional information about this page in bits 0-8 and destroys the previous contents of the AC. The format of the information placed in the specified AC is as follows:



| Bits | Name | Contents |
|---|---|---|
| 0 | WP | The write protect bit for the logical page which corresponds to the physical page specified by bits 8-15. |
| 1-3 | Map | The map which was used to perform the translation between logical page number and physical page number is as follows:<br>000 User A<br>001 Reserved for future use.<br>010 User B<br>011 Reserved for future use<br>100 Data channel A<br>101 Data channel C<br>110 Data channel B<br>111 Data channel D |
| 4-5 | --- | Reserved for future use. |
| 6-8 | Validity | If these bits are 1, and bit 0 is a 1, the logical page which corresponds to the physical page specified by bits 9-15 is validity protected. |
| 9-15 | Physical Page | The number of the physical page which corresponds to the logical page given in the preceding INITIATE PAGE CHECK instruction. |

## Map Supervisor Page 31

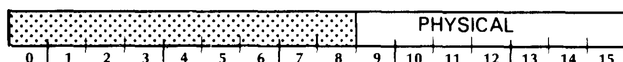
**DOB** *ac,MAP*




Specifies the physical page corresponding to logical page 31 of the supervisor's address space. It is used by the supervisor to access a user's memory space when many references will be required.

Bits 9-15 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 supervisor mode.

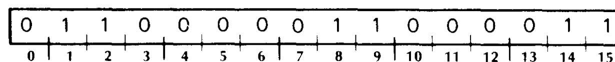
The contents of the specified AC remain unchanged. The format of the specified AC is as follows:




| Bits | Name | Contents |
|---|---|---|
| 0-8 | --- | Reserved for future use. |
| 9-15 | Physical Page | The number of the physical page to which logical page 31 should be mapped when in supervisor mode. |

## Map Single Cycle

**NIOP MAP**




Maps one memory reference using the last user map. It is used by the supervisor to access a user's memory space when only one or two references are required.

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

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

# FLOATING POINT ARITHMETIC

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

If the floating point instruction set is not installed, floating point instructions will be executed as NO OPS, i.e., JMP .+1 or EJMP .+1 as appropriate.

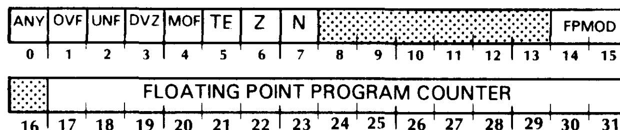
**NOTE** *The S/130 floating point instruction set expects normalized input numbers. If normalized input is not provided, the results will be undefined.*

## 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.




| Bits | Name | Contents |
|---|---|---|
| 0 | ANY | Indicates that any of bits 1-4 are set |
| 1 | OVF | Overflow Indicator--during processing of a floating point number, an exponent overflow occured; the result is correct except the exponent is 128 too small. |
| 2 | UNF | Underflow Indicator--during the processing of a floating point number, an exponent underflow occured; the result is correct except that the exponent is 128 too large. |
| 3 | DVZ | Divide by Zero--during the processing of a floating point number, a zero divisor was detected; division was aborted and the operands remain unchanged. |
| 4 | MOV | Mantissa Overflow--during a floating point FSCAL instruction, a significant bit was shifted out of the high order end of the mantissa; also, this bit is set, during a FIX instruction, the result cannot fit into the destination location. |
| 5 | TE | Trap Enable--If this bit is 1, the setting of any of bits 1-4 will result in a floating point fault. |
| 6 | Z | Zero bit--The result of the last floating point operation was equal to zero. |
| 7 | N | Negative bit--The result of the 1st floating point operation was less than zero. |
| 8-13 | RES | Reserved for future use. |
| 14-15 | FPMOD | Indicates computer series supporting the floating point instruction set:<br>00 S/200, C/300 Series<br>01 S/130 Series<br>10 Reserved for future use<br>11 Reserved for future use |
| 16 | RES | 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. |

## 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. See the description of the floating point status register above for more detail on them.

## 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:

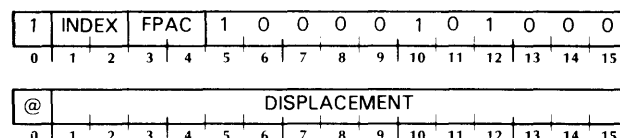
| WORD PUSHED | DESCRIPTION |
|---|---|
| 1 | AC0 |
| 2 | AC1 |
| 3 | AC2 |
| 4 | AC3 |
| 5 | Bit 0: Carry; and<br>Bits 1-15: return address |

**NOTES** *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. The fault 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.*

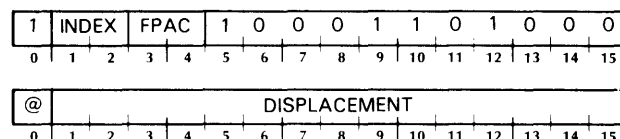
## Load Floating Point Single

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




## Load Floating Point Double

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

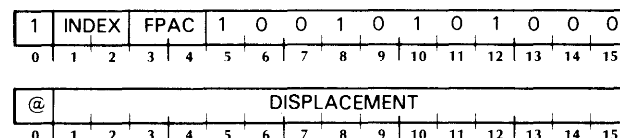



Moves a word out of memory into a specified FPAC.

Computes the effective address *E* and places the 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 bits in the FPSR to reflect the new contents of FPAC. For single precision, the low-order 32 bits of FPAC are set to 0.

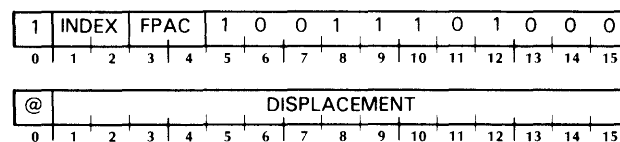
## Store Floating Point Single

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




## Store Floating Point Double

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

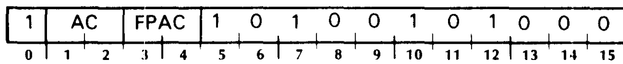



Stores the contents of a specified FPAC into a memory location.

Computes the effective address *E* and places the floating point number contained in FPAC in memory beginning at the location addressed by E. Destroys the previous contents of the addresed 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.

## Float From AC

**FLAS** *ac,fpac*

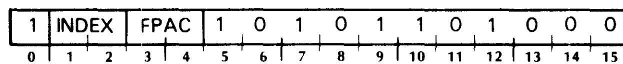


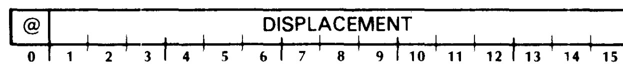
Converts the signed two's complement number contained in the specified accumulator to a single precision floating point number, places the result in the specified FPAC, and sets the low-order 32 bits of the FPAC to 0. Leaves the contents of the specified accumulator unchanged and destroys the previous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

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

## Float From Memory

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





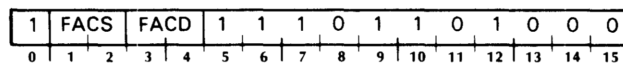
Converts the contents of two memory locations to floating point format and places the result in a specified FPAC.

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

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

## Move Floating Point

**FMOV** *facs,facd*

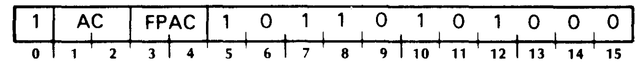


Moves the contents of one FPAC to another FPAC.

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 bits in the FPSR are set to reflect the new contents of FACD.

## Fix To AC

**FFAS** *ac,fpac*



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

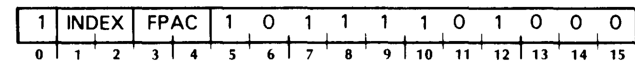
Forms the absolute value of the integer portion of the floating point number in FPAC. Extracts the 15 least significant bits from this value and, if the number in FPAC is negative, forms the two's complement of the integer. Then places the result in the specified accumulator, sets the Z and N bits in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

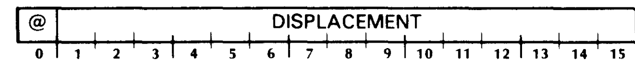
If the number in FPAC is less than $-32{,}767_{10}$ or greater than $+32{,}767_{10}$, this instruction sets the MOF bit in the floating point status register to 1.

**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]*





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

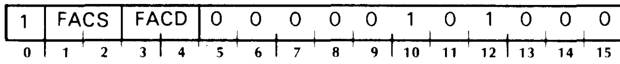
Forms the absolute value of the integer portion of the floating point number in FPAC. Extracts the 31 least significant bits from this value and, if the number in FPAC is negative, forms the two's complement of the integer. Then places the result into the location addressed by E, sets the Z and N bits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

If the number in FPAC is less than $-2{,}147{,}483{,}647_{10}$ or greater than $+2{,}147{,}483{,}647_{10}$, this instruction sets the MOF bit in the floating point status register to 1.

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

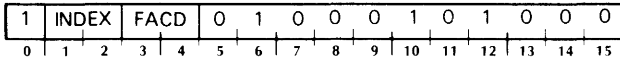
## Add Single (FPAC to FPAC)

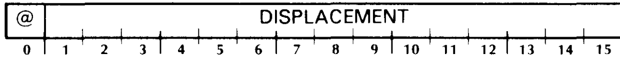
**FAS** *facs,facd*



## Add Single (memory to FPAC)

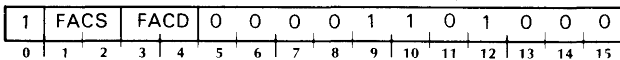
**FAMS** *facd,[@]displacement[,index]*





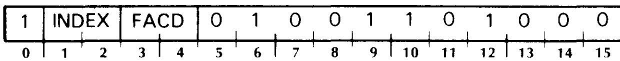
## Add Double (FPAC to FPAC)

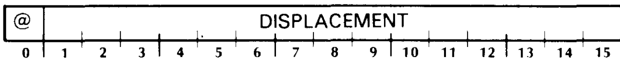
**FAD** *facs,facd*



## Add Double (memory to FPAC)

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





Adds the floating point number in the source location to the floating point number in FACD and places the normalized result in FACD. Destroys the previous contents of FACD, leaves the contents of the source location unchanged and updates the Z and N bits in the FPSR to reflect the new contents of FACD.

For an add from memory, the effective address *E* is computed. E addresses either a 2-word (single precision) or 4-word double precision) operand.

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.

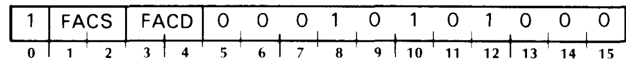
After alignment, the mantissas are added together. The result of this addition is termed the intermediate result. One guard digit is provided for the intermediate result, which is used if normalization is required. The sign of the intermediate result is determined from the signs of the two operands by 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 FPSR, and the number in FACD is correct except that the exponent is 128 too small.

If there is no mantissa overflow, the mantissa of the intermediate result is examined for leading hex 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 FPSR and the instruction is terminated. The number in the FACD is correct except that the exponent is 128 too large.

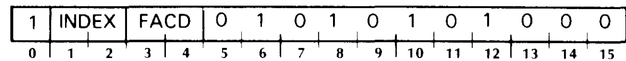
## Subtract Single (FPAC from FPAC)

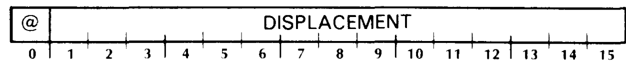
**FSS** *facs,facd*



## Subtract Single (memory from FPAC)

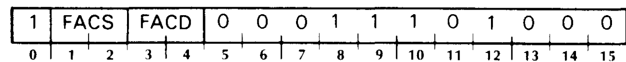
**FSMS** *facd,[@]displacement[,index]*





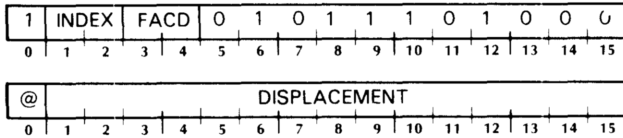
## Subtract Double (FPAC from FPAC)

**FSD** *facs,facd*



## Subtract Double (memory from FPAC)

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



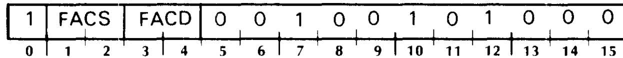

Subtracts the floating point number in the source location from the floating point number in FACD and places the normalized result in the FACD. Destroys the previous contents of FACD, leaves the contents of the source location unchanged, and updates the Z and N bits in the FPSR to reflect the new contents of FACD.

For a subtract from memory, the effective address *E* is computed. E addresses either a 2-word (single precision) or 4-word (double precision) operand.

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.

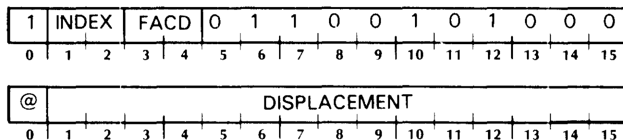
## Multiply Single (FPAC by FPAC)

**FMS** *facs,facd*



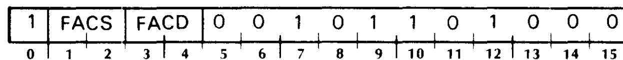

## Multiply Single (FPAC by memory)

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



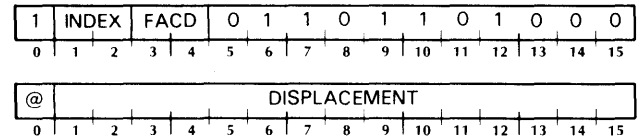

## Multiply Double (FPAC by FPAC)

**FMD** *facs,facd*




## Multiply Double (FPAC by memory)

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




Multiplies the floating point number in FACD by the floating point number in the source location and places the normalized result in FACD. Destroys the previous contents of FACD, leaves the contents of the source location unchanged, and updates the Z and N bits in the FPSR are set to reflect the new contents of FACD.

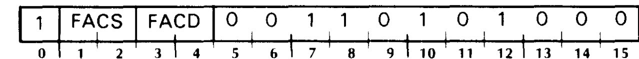
For a multiply from memory, the effective address *E* is computed. E addresses either a 2-word (single precision) or 4-word (double precision) operand.

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 bit in the FPSR is set. The number is correct except that, for exponent overflow, the exponent is 128 too small, and for exponent underflow, the exponent is 128 too large.

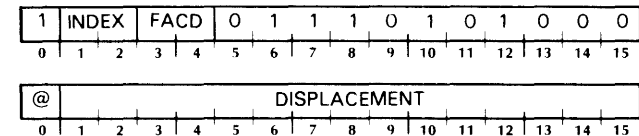
## Divide Single (FPAC by FPAC)

**FDS** *facs,facd*



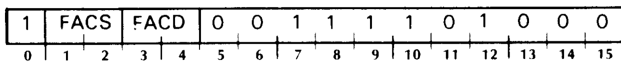

## Divide Single (FPAC by memory)

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



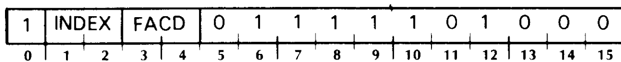
## Divide Double (FPAC by FPAC)

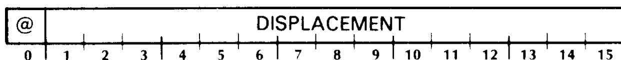
**FDD** *facs,facd*




## Divide Double (FPAC by memory)

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




Divides the floating point number in FACD by the floating point number in the source location and places the normalized result in FACD. Destroys the previous contents of FACD, leaves the contents of the source location unchanged, and updates the Z and N bits in the FPSR to reflect the new contents of FACD.

For a divide from memory, the effective address *E* is computed. E addresses either a 2-word (single precision) or 4-word (double precision) operand.

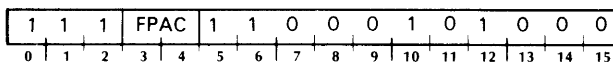
The source operand is checked for a zero mantissa. If the mantissa is zero, the DVZ bit is set in the FPSR and the instruction is terminated. The number in FACD remains unchanged. If the mantissa is nonzero, the previous contents of FACD are lost. The two mantissas are compared and if the mantissa of 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 FPSR is set. The number in FACD is correct except that, for exponent overflow, the exponent is 128 too small, and for exponent underflow, the exponent is 128 too large.

## Negate

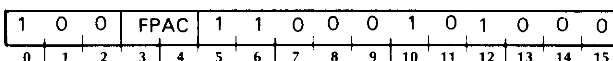
**FNEG** *fpac*




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

## Normalize

**FNOM** *fpac*

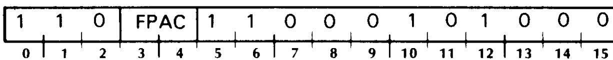



Normalizes the floating point numbers in FPAC. Sets a true zero in FPAC if all the bits of the mantissa are zero. Sets the UNF bit in the FPSR if an exponent underflow occurs. The number in FPAC is then correct, except that the exponent is 128 too large.

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

## Absolute Value

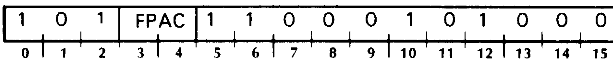
**FAB** *fpac*




Sets the sign bit of FPAC to 0. Also sets the exponent to zero if the mantissa is zero. Leaves bits 1-63 of FPAC unchanged, and updates the Z and N bits in the FPSR to reflect the new contents of FPAC.

## Read High Word

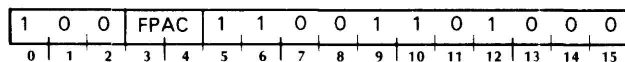
**FRH** *fpac*




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 bits in the FPSR.

## Scale

**FSCAL** *fpac*

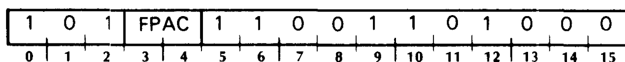


Shifts the mantissa of the floating point number in FPAC either right or left, depending upon the contents of bits 1-7 of AC0. Leaves the contents of AC0 unchanged.

Bits 1-7 of AC0 are treated as an exponent in *Excess 64* representation. The difference between this exponent and the exponent in FPAC is computed by subtracting the exponent in FPAC from the number contained in AC0 bits 1-7. If the difference is zero, the instruction is terminated. If the difference is positive, the mantissa contained in FPAC is shifted right that number of hex digits. If the difference is negative, the mantissa contained in FPAC is shifted left that number of hex digits and if bits are lost the MOF bit in the FPSR is set. After the shift, the contents of bits 1-7 of AC0 replace the exponent contained in FPAC. Bits shifted out of either end of the mantissa are lost. If the entire mantissa is shifted out of FPAC, FPAC is set to true zero. The Z and N bits in the FPSR are set to reflect the new contents of FPAC.

## Load Exponent

**FEXP** *fpac*

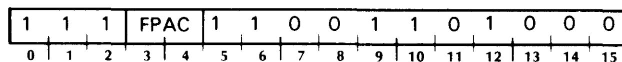


Places bits 1-7 of AC0 in bits 1-7 of the specified FPAC. Ignores bits 0 and 8-15 of AC0. Leaves unchanged bits 0 and 8-63 of FPAC and the entire contents of AC0. Also sets bits 0-7 (the sign and exponent) to zero if bits 8-63 (the mantissa) of FPAC are zero. Leaves bits 1-7 of FPAC unchanged if FPAC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

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

## Halve

**FHLV** *fpac*

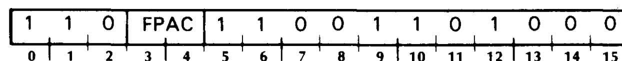


Shifts the mantissa contained in FPAC right one bit position, fills the vacated bit position with a zero and places the bit shifted out in the guard digit. Then normalizes the number and places the result in FPAC. Sets the UNF bit in the FPSR if the normalization process causes an exponent underflow.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

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

## Intergerize

**FINT** *fpac*

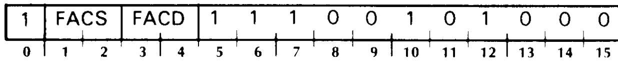


Sets the fractional portion of the floating point number in the specified FPAC to zero and normalizes the result. Updates the Z and N bits in the FPSR to reflect the new contents of the specified FPAC.

> **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.*

## Compare Floating Point

**FCMP** *facs,facd*




Compares two floating point numbers and sets the Z and N bits in the FPSR accordingly.

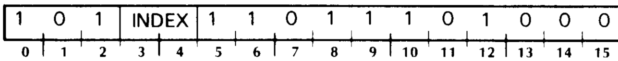
Algebraically compares the floating point numbers in FACS and FACD to each other and updates the Z and N bits in the FPSR to reflect the result. Leaves the contents of FACS and FACD unchanged. The results of the compare and the corresponding bit settings are shown in the table below.

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

**NOTE** *Unnormalized operands give unspecified results.*

## Load Floating Point Status

**FLST** *[@]displacement[,index]*

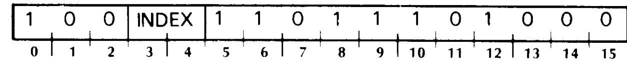


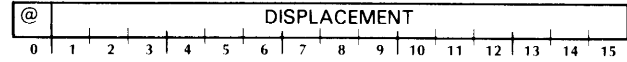
Moves the contents of two specified memory locations to the FPSR.

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

## Store Floating Point Status

**FSST** *[@]displacement[,index]*



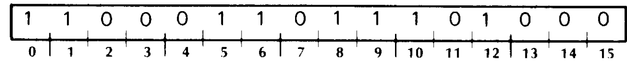




Moves the contents of the FPSR to two specified memory locations.

Computes the effective address *E*, and places the 32-bit contents of the FPSR in the memory location addressed by E, leaving the contents of the FPSR unchanged.

## Trap Enable

**FTE**

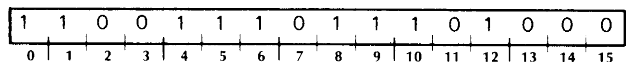



Sets the trap enable bit of the FPSR to 1.

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

## Trap Disable

**FTD**

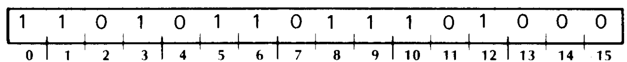



Sets the trap enable bit of the FPSR to 0.

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

## Clear Errors

**FCLE**

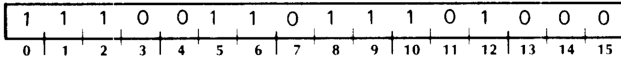



Sets bits 0-4 of the FPSR to 0.

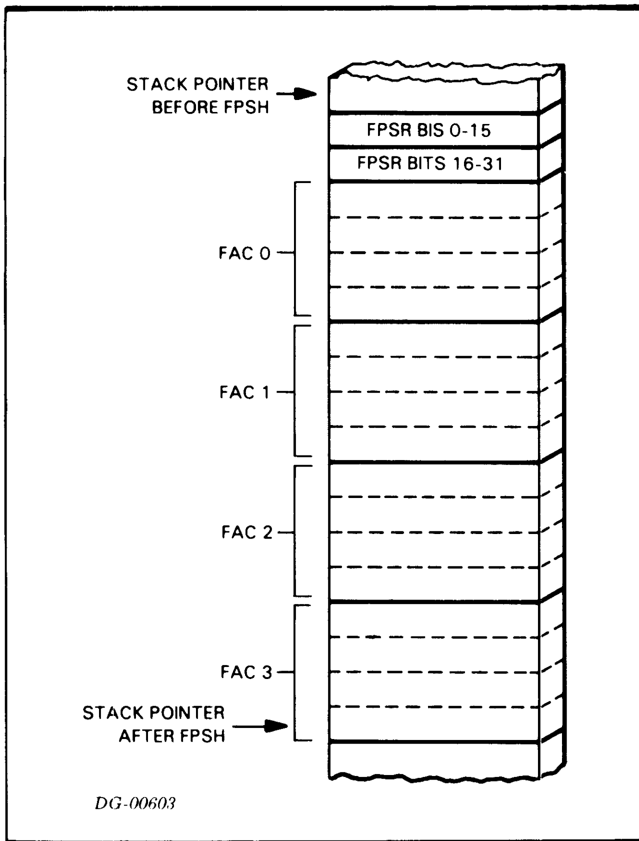
**NOTE** *The I/O RESET instruction will set these bits to 0.*

## Push Floating Point State

### FPSH

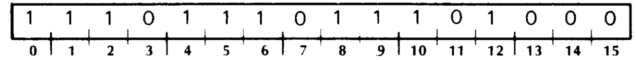



Pushes an 18-word floating point return block onto the user stack, leaving the contents of the floating point accumulators and the FPSR unchanged. The format of the 18 words pushed is as follows:

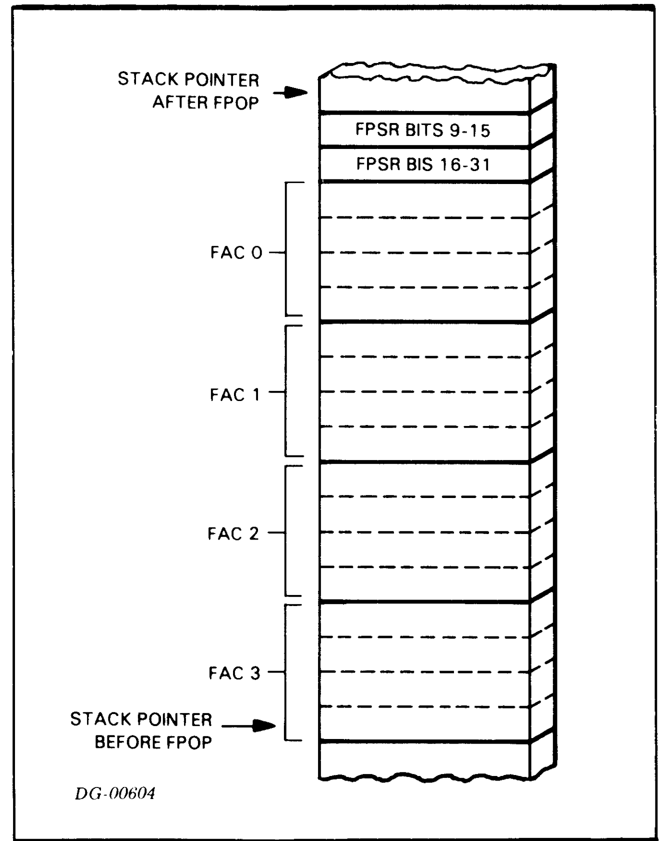



## Pop Floating Point State

### FPOP




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



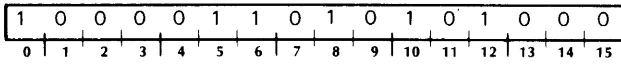

**NOTE** *Because of the potentially long time required to perform some floating point instructions in relation to I/O interrupt requests, these instructions are interruptable. Because the FACD, stack pointer, and program counter are not updated until the completion of these instructions, any interrupt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

## Arithmetic Test

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

## No Skip

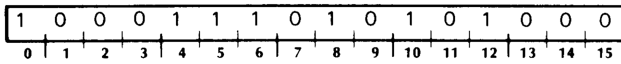
### FNS



The next sequential word is executed.

## Skip Always

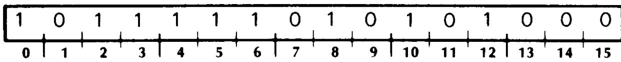
### FSA



The next sequential word is skipped.

## Skip On Greater Than Zero

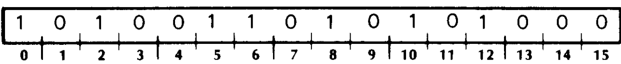
### FSGT



Skips the next sequential word if both the Z and N bits of the FPSR are 0.

## Skip On Less Than Zero

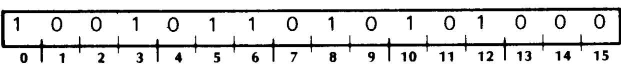
### FSLT



Skips the next sequential word if the N bit of the FPSR is 1.

## Skip On Zero

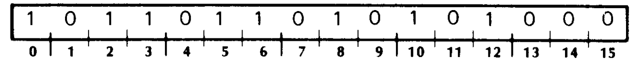
### FSEQ



Skips the next sequential word if the Z bit on the FPSR is 1.

## Skip On Less Than Or Equal To Zero

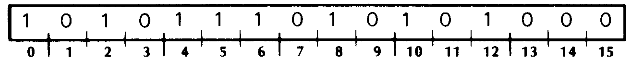
### FSLE



Skips the next sequential instruction if either the Z bit or the N bit of the FPSR is 1.

## Skip On Greater Than Or Equal To Zero

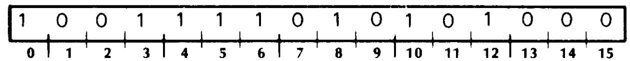
### FSGE



Skips the next sequential word if the N bit of the FPSR is 0.

## Skip On Non-Zero

### FSNE



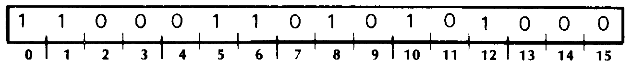
Skips the next sequential word if the Z bit of the FPSR is 0.

## Error Test

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

## Skip On No Mantissa Overflow

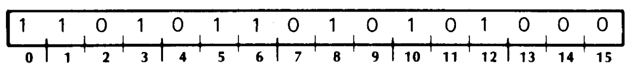
### FSNM



Skips the next sequential word if the mantissa overflow (MOF) bit of the FPSR is 0.

## Skip On No Underflow

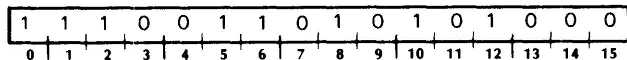
### FSNU



Skips the next sequential word if the underflow (UNF) bit of the FPSR is 0.

## Skip On No Overflow

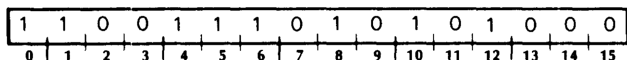
### FSNO



Skips the next sequential word if the overflow (OVF) bit of the FPSR is 0.

## Skip On No Zero Divide

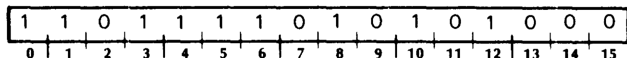
### FSND



Skips the next sequential word if the divide by zero (DVZ) bit of the FPSR is 0.

## Skip On No Underflow And No Zero Divide

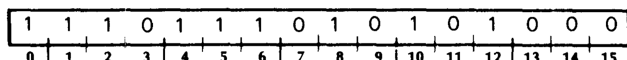
### FSNUD



Skips the next sequential word if both the underflow (UNF) bit and the divide by zero (DVZ) bit of the FPSR are 0.

## Skip On No Overflow and No Zero Divide

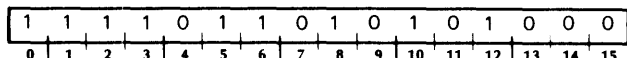
### FSNOD



Skips the next sequential word if both the overflow (OVF) bit and the divide by zero (DVZ) bit of the FPSR are 0.

## Skip On No Underflow And No Overflow

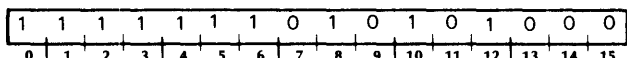
### FSNUO



Skips the next sequential word if both the underflow (UNF) bit and overflow (OVF) bit of the FPSR are 0.

## Skip On No Error

### FSNER



Skips the next sequential word if bits 1-4 of the FPSR are all 0.

# CHARACTER INSTRUCTION SET

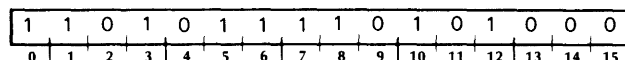
The character instruction set performs operations on strings of bytes.

**NOTE** *Because of the potentially long time that may be required to complete any instruction in the character set (relative to I/O requests), all instructions in this set are interruptable. If an instruction is interrupted, the program counter is decremented by one before it is placed in location 0 so that it points to the instruction in progress. All the instructions maintain their operands in such an manner that any interrupt service routine that returns control to the interruupted program via the address stored in memory location 0 will correctly restart the interrupted instruction.*

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

## Character Move

### CMV



Moves a string of bytes from one area of memory to another under control of the values in the four accumulators.

Fetching and storing may proceed from right to left or from left to right and may be in opposite directions. Moving continues until the destination field is filled. If the source field is longer than the destination field the carry bit is set to 1, otherwise it is set to 0. If the source field is shorter than the destination field, the destination field is padded with space characters.

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

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

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

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

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

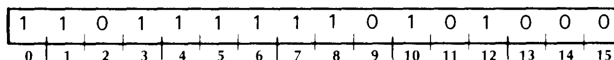
Upon termination, AC0 contains 0; AC1 contains the number of source bytes remaining to be fetched; AC2 contains a byte pointer which is the address of the next byte after the destination field; and AC3 contains a byte pointer which is the address of the next byte to be fetched.

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

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

## Character Compare

### CMP




Compares two strings of bytes and returns a code reflecting the results in AC1.

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

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

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

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

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

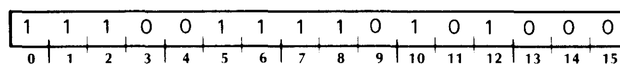
AC3 must contain a byte pointer which is the address of the first byte to be processed in string 1.

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

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

## Character Translate

### CTR




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.

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

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

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

| 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 |

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

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

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

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

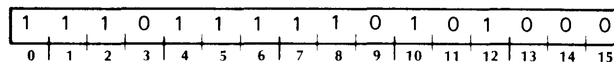
The fields may overlap in any way. However, processing is done one character at a time, so unusual side effects may be produced by certain types of overlap.

Upon termination of the instruction with the move option, AC0 contains the resolved address of the word which contains the byte pointer to the translation table; AC1 contains 0; AC2 contains a byte pointer which is the address of the next byte after string 2; and AC3 contains a byte pointer which is the address of the next byte after string 1.

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

## Character Move Until True

### CMT




Moves a string of bytes from one area of memory to another until either a table-specified delimiter character is encountered or the source string is exhausted.

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

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

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

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

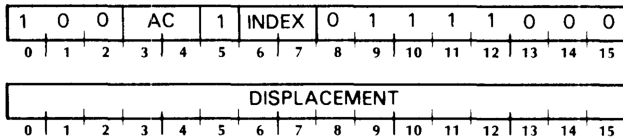
AC3 must contain a byte pointer which is the address of the first byte to be processed in the source string.

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

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

## Extended Load Byte

**ELDB** *ac,displacement[,index]*



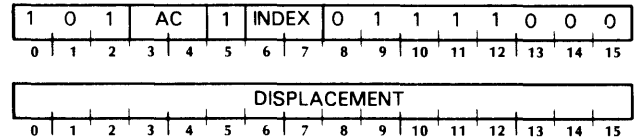

A byte pointer is formed by taking the index value, multiplying it by 2, and adding the low-order 16 bits of the result to the displacement. The byte addressed by this byte pointer is placed in bits 8-15 of the specified AC. Bits 0-7 of the specified AC are set to 0. Neither the index value nor the displacement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

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

| INDEX BITS | INDEX |
|---|---|
| 00 | 0 |
| 01 | Address of the displacement filed |
| 10 | Contents of AC2 |
| 11 | Contents of AC3 |

## Extended Store Byte

**ESTB** *ac,displacement[,index]*




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

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

| INDEX BITS | INDEX |
|---|---|
| 00 | 0 |
| 01 | Address of the displacement filed |
| 10 | Contents of AC2 |
| 11 | Contents of AC3 |

## Programming Examples

The following sequence of instructions will reverse the order of a string of bytes contained at **SRC**, and store the result at **DST**.

```
        LDA     1,C100  ;Load length of source
        NEG     1,0     ;Negative length for destination
        LEF     3,SRC   ;Get address of source
        MOVZL   3,3     ;Form bype pointer
        LEF     2,DST+37 ;Get address of last
                        ; destination word
        MOVOL   2,2     ;Form byte pointer
        CMV             ;Do it
        ...

C100:   100             ;Byte count
SRC:    .BLK 40         ;Area for 64 source bytes
DST:    .BLK 40         ;And 64 destination bytes
```

The next example can be used by a program to determine whether or not it is running in a processor which supports the character instruction set. It makes use of the fact that **ELDB**, which is a two-word instruction, is treated as a one-word no-op by processors that do not support the character instructions. In this case the displacement, which is in the word following **ELDB**, will be executed as an instruction. The displacement of $402_8$ is equivalent to a **JMP** .+2 instruction.

If the **ELDB** is supported, $402_8$ is interpreted as a pointer to the high-order byte of location $201_8$. This byte is loaded into AC0 and execution is transferred to a location named **HAVCH**. Note that if this happens, the contents of AC0 are lost.

```
        ELDB    0,402   ;Displacement goes in next word
        (JMP    .+2)    ;402 = JMP
        JMP     HAVCH   ;Here if ELDB worked
        ...             ;Here if it didn't
```

# WRITEABLE CONTROL STORE

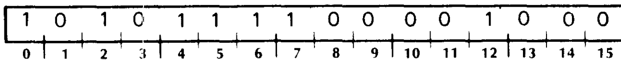
The writeable control store (WCS) allows the user to transfer control to any one of 16 entry points in WCS. With these routines the full power of the microcode processor can be used.

## Placing Microcode In WCS

Before the user can use the XOP feature to execute instructions in WCS, the microcode must be placed in the WCS locations. This discussion treats only how to place microcode in WCS. For a detailed discussion of how to write microprograms see *Microprogramming ECLIPSE Computers with the WCS Feature (DGC No. 015-000069)*.

## Load Control Store Formatted

### LCSF




AC1 contains a count of the number of microinstructions to be loaded. The address of the source location in main memory is contained in bits 1-15 of AC2. The address of the destination location, or control store address, is contained in bits 4-15 of AC3. If bit 0 of either AC2 or AC3 is 1, it is assumed that the address contained in bits 1-15 is an indirect address in main memory. Before the data movement occurs, the indirection chain is followed and the resultant effective address is placed in the accumulator. The LCSF instruction moves words sequentially from memory to the control store. Each bit is complemented as it is moved.

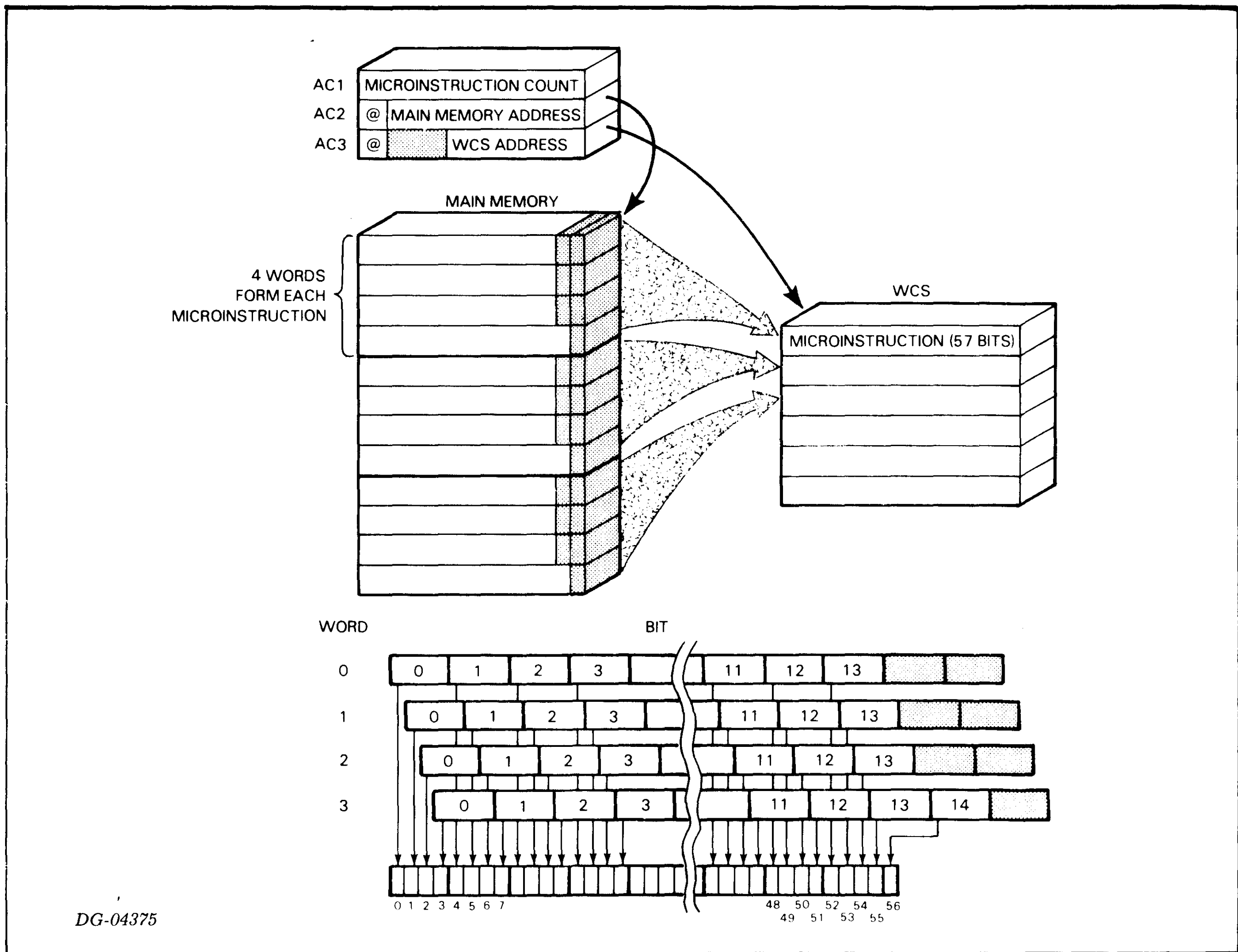
The final direct address in AC3 addresses a location in the WCS memory. The WCS starts at microaddress $4000_8$ and extends to $5777_8$. Since microinstructions require 57 bits, four 16-bit words are required for each microcode word. The contents of these four 16-bit words are entered in the microcode word as follows:

The complements of bits 0 through 13 of the first word in main memory are loaded into bits 0, 4, 8, 12, 16, 20, 24, 28, 32, 36, 40, 44, 48 and 52 of the first microcode word.

The complements of bits 0 through 13 of the second word in main memory are loaded into bits 1, 5, 9, 13, 17, 21, 25, 29, 33, 37, 41, 45, 49 and 53 of the first microcode word.

The complements of bits 0 through 13 of the third word in main memory are loaded into bits 2, 6, 10, 14, 18, 22, 26, 30, 34, 38, 42, 46, 50 and 54 of the first microcode word.

The complements of bits 0 through 14 of the fourth word in main memory are loaded into bits 3, 7, 11, 15, 19, 23, 27, 31, 35, 39, 43, 47, 51, 55 and 56 of the first microcode word.




The next four words in main memory similarly are loaded into the second microcode word, and so on.

For each microcode word move, the count in AC1 is decremented by one, the source address in AC3 is incremented by four, and the destination address in AC3 is incremented by one. Upon completion of the instruction, AC1 contains zero, 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 source address after $77777_8$ is 0. If at any the time the destination address is outside the range of $4000_8$ to $5777_8$, the instruction is terminated.

**NOTES** *Because of the potentially long time that may be required to perform this instruction in relation to I/O requests, this instruction is interruptable. If an LCSF instruction is interrupted, program counter is decremented by one before it is placed in location 0 so that it points to the LCSF instruction. Because the addresses and the word count are updated after every word stored, an interrupt service routine that returns control to the interrupted program via the address stored in memory location 0 will correctly restart the LCSF instruction.*

*The LCSF instruction can be disabled for any user by enabling I/O protection for that user with the MAP. An attempt by this user to use the LCSF instruction will cause a protection fault.*

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

## Enter WCS

**XOP1** *acs,acd,entry number*

The microprogram in WCS whose entry number corresponds to bits 6-9 in the ENTER WCS instruction is executed. Permissible entry numbers are 0-15, corresponding to the first 16 locations in WCS. The executed microprogram controls the use of accumulators, whether or not they are changed, and the location of the next instruction.

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

# CHAPTER V
# INPUT/OUTPUT

## INTRODUCTION

This chapter describes the Input/Output (I/O) instruction set in the S/130. The I/O instructions control the operation of the I/O devices connected to the computer. We will first discuss the operation of the I/O system, and then the individual instructions. The I/O interrupt system and the operation of the *VECTOR* instruction are discussed separately in later sections of the chapter.

## THE S/130 I/O SYSTEM

The S/130 can communicate with I/O devices using two methods: programmed I/O and data channel I/O. Programmed I/O is used to transfer data to and from all slow I/O devices, such as terminals, and to control data channel devices. Only certain fast devices, such as discs and tape units, use data channel I/O.

### Programmed I/O

Programmed I/O transfers data one byte 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.

## Device Codes

The S/130 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 only will 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.

## Busy and Done Flags

I/O devices are controlled by manipulating their Busy and Done flags (but note that data channel devices require several full 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 mnemonics appended to the instruction, corresponding to a code in bits 8 and 9 of the I/O instruction format. 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.

## Data Channel

Data channel devices are controlled in two phases. Phase I specifies the parameters of the transfer, i.e., the starting location in memory and the number of words to be transferred. This is done with programmed I/O instructions. Phase II consists of either a Read or a Write command, which are flag commands similar to those discussed above. nce the flag command is issued, the data transfer takes place when both the data channel device and the processor are ready. o 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 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.

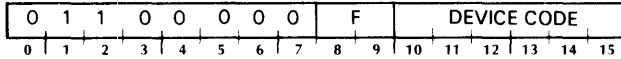
After handling all data channel requests, the processor then handles all outstanding I/O interrupt requests. Only then does program execution continue.

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 INSTRUCTIONS

## No I/O Transfer

**NIO** *[f]* *device*

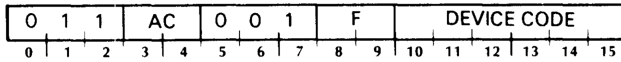


Used when a Busy or Done flag must be changed with no other operation taking place.

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

## Data In A

**DIA** *[f]* *ac,device*



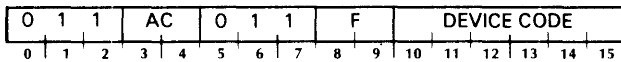
**Transfers data from the A buffer of an I/O device to an accumulator.**

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

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

## Data In B

**DIB** *[f]* *ac,device*



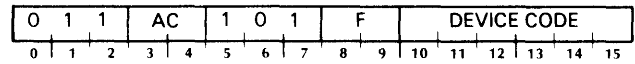
**Transfers data from the B buffer of an I/O device to an accumulator.**

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

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

## Data In C

**DIC** *[f]* *ac,device*



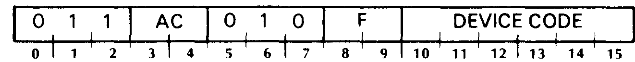
**Transfers data from the C buffer of an I/O device to an accumulator.**

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

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

## Data Out A

**DOA** *[f]* *ac,device*



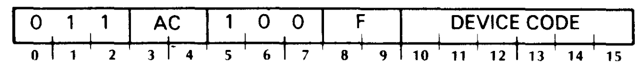
**Transfers data from an accumulator to the A buffer of an I/O device.**

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

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

## Data Out B

**DOB** *[f]* *ac,device*



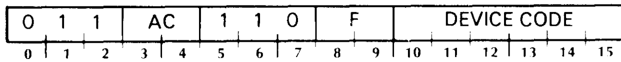
**Transfers data from an accumulator to the B buffer of an I/O device.**

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

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

## Data Out C

**DOC** *[f]* *ac,device*



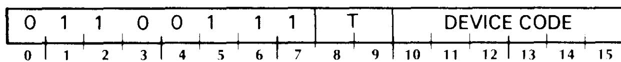

Transfers data from an accumulator to the C buffer of an I/O device.

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

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

## I/O Skip

**SKP** *[t]* *device*




Tests a flag and skips the next instructions if the test condition is true.

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

## Programming Example

In a system with a full-duplex terminal, the program must *echo* any characters it reads so that the user can see what he has typed. The routine below uses programmed I/O instructions to read a character from the terminal and echo it back. TTI is the mnemonic for the keyboard, and TTO is the mnemonic for the printer or display. The character is in AC0 at the end of the routine.

```
...
...
SKPDN   TTI     ;Character typed yet?
JMP     .-1     ;No, go back and try again
DIAC    0,TTI   ;Yes, read it into AC0 and clear done
SKPDN   TTO     ;Can we echo it?
JMP     .-1     ;No, wait until printer is ready
DOAS    0,TTO   ;Yes, send character and start printer
...
```

See *Programmer's Reference Manual - Peripherals (DGC No. 015-000021)* for a more complete set of examples, including some data channel examples.

## I/O Interrupts

The I/O interrupt system in the S/130 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, permitting 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.

**Non-base 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.

## Processing an Interrupt

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. If 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 to 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 to location 0 will return 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 it might be reasonable for a disc to interrupt a teletype, the inverse certainly would not be true. 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 crude 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 S/130 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 interruptable 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.

## Priority Interrupt Handler

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. See the *Vector* instruction section for more detail on this.

# INTERRUPT INSTRUCTIONS

All the interrupt instructions use the device code $77_8$. The assembler recognizes the mnemonic CPU for this device code. See Chapter II for detailed information on I/O instruction formatting.

**ASSEMBLER CONVENTIONS**

Many of these instructions have special mnemonics which can be used in place of the standard mnemonics. The one limitation is that the mnemonics for controlling the state of the Interrupt On flag cannot be appended to the 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

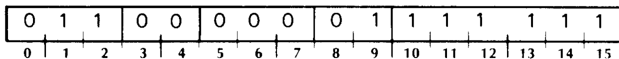
**MSKO** *ac.*

The special mnemonic sets bits 8 and 9 to 00. The special mnemonics are given first below, followed by the standard form.

## Interrupt Enable

**INTEN**

**NIOS CPU**



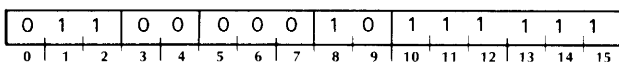
Sets Interrupt On flag to 1.

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

## Interrupt Disable

**INTDS**

**NIOC CPU**

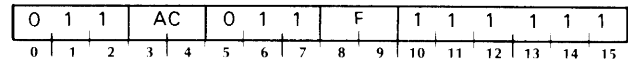


Sets Interrupt On flag to 0.

## Interrupt Acknowledge

**INTA**

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



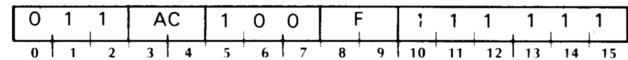
Returns device code of an interrupting device.

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

## Mask Out

**MSKO**

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



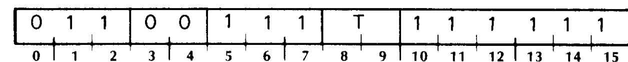
Changes the priority mask.

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

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

## CPU Skip

**SKP** *[t]* **CPU**



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.

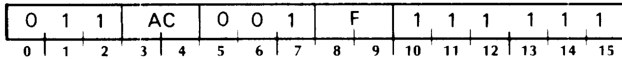
## SPECIAL CENTRAL PROCESSOR INSTRUCTIONS

This group of instructions perform special functions for the CPU.

### Read Switches

**READS** *ac*

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



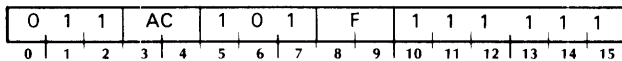

Places the contents of the console switches into an accumulator.

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

### Reset

**IORST**

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




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

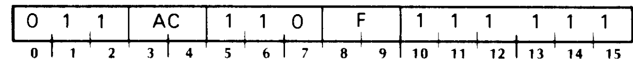
The Busy and Done flags in all I/O devices are set to 0. The 16-bit priority mask is set to 0. The Interrupt On flag is set according to the function specified by F.

> **NOTES** *The assembler recognizes the mnemonic IORST as equivalent to the instruction* **DICC 0,CPU.**
>
> *If the mnemonic DIC is used to perform this function, an accumulator must be coded to avoid assembly errors. Regardless of how the instruction is coded, during execution, the AC field is ignored and the contents of the AC remain unchanged.*

### Halt

**HALTA** *ac*

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




Stops the processor.

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

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

## USE OF 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. This section will describe the operation and use of the *Vector* instruction.

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.

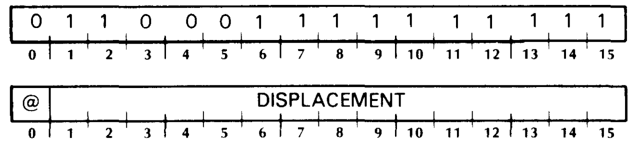
Modes D and E both initiate a new stack. You should use them only when operating at base level (no interrupt processing in progress) since they set up a new vector stack for use by the interrupt handler and store the (old) user stack parameters in it. Once this new stack has been set up, there is no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

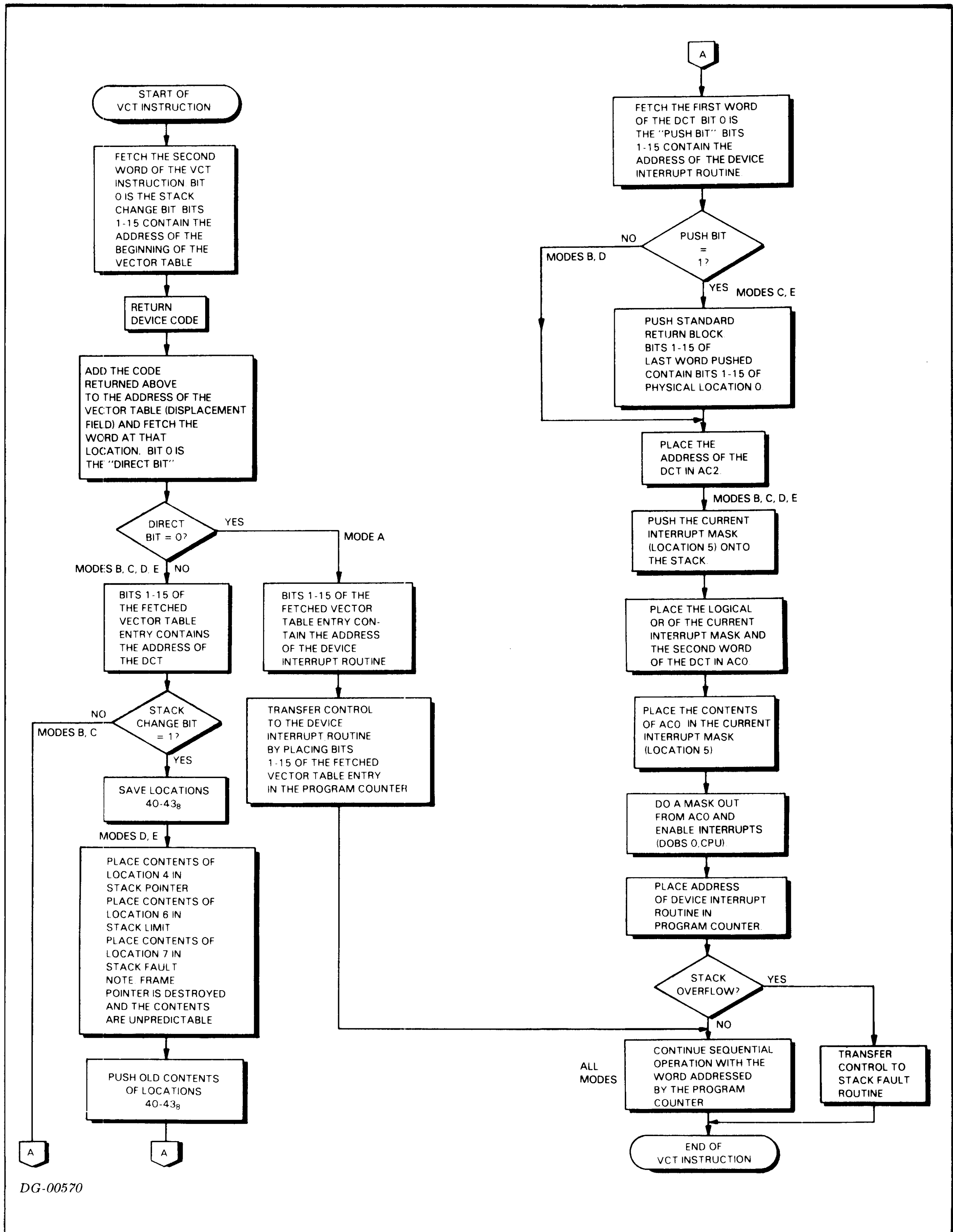
Note that using the faster modes gives you faster reaction time at the interrupt, but requires more careful design of the interrupt handler that eventually receives control. The interrupt handler must do what the *Vector* instruction did not take the time to do, i.e., store the contents of the carry bit and any accumulators it uses when servicing the interrupt. There are no problems doing this when using mode A, but mode B uses AC0, destroying the previous contents.

### Vector On Interrupting Device Code

**VCT** *[@]displacement*




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.



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
TRANSFER CONTROL TO THE DEVICE INTERRUPT ROUTINE BY PLACING BITS 1-15 OF THE FETCHED VECTOR TABLE ENTRY IN THE PROGRAM COUNTER
SAVE LOCATIONS $40\text{-}43_8$
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 $40\text{-}43_8$
A
A
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

The accompanying flow chart 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 state of these bits collectively determine which mode will be used by the *Vector* instruction. This relationship is as follows:

| Direct | Stack Change | Push | Mode |
|---|---|---|---|
| 0 | don't care (d.c.) | d.c. | 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 the process through 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.

### 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 contruct 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 4 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 $40\text{-}43_8$ (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. Interrupts are enabled and control is passed to the selected device interrupt handler.

## Programming Example

The following example illustrates the use of the *Vector* instruction. This example assumes a system with only three peripherals: an event counter that requires mode A service; a slow speed input device (TTY input); and a slow speed output device (TTY output). The output device is of lower priority than the input device.

```
                          ;USE OF ECLIPSE
                          ;VECTOR INSTRUCTION

         .LOC 0           ;Start assembly at 0
INTR:    0                ;Interrupt return
INTE:    PI               ;Address of program
                          ;interrupt routine
SC:      SCH              ;Address of SCL handler
PF:      PFH              ;Address of PF handler
VSP:     VS               ;Vector stack pointer
CURMK:   0                ;Current mask
VSL:     VL               ;Vector stack limit
VSF:     VF               ;Vector stack fault handler


         .LOC    50       ;Next location is 50 octal
LEVEL:   -1               ;Interrupt level count
         .LOC    1000     ;Next location is 1000 octal
PI:      ISZ     LEVEL    ;Base level?
         JMP     .+3      ;No
         VCT     @VTAB    ;Base level vector--
                          ;   Sets stack change bit to 1 @
         VCT     VTAB     ;Non-base level vector
                          ;Stack change bit is 0

DISMIS:  POP     3,3      ;Pop old mask into AC3
         DOBC    3,CPU    ;Maskout, disable interrupts
         STA     3,CURMK  ;Store mask into current mask
         LDA     3,LEVEL  ;Pick up level
         SBI     1,3      ;Subtract 1
         STA     3,LEVEL  ;Store it back
         COM#    3,3,SZR  ;Base level?
         JMP     .+3      ;No--just return
         INTEN            ;Yes
         RSTR             ;Restore
         INTEN            ;Interrupt enable
         POPB             ;Return
```

```
VTAB:    @ SPUR                 ;SPUR is address of DCT
         @ SPUR                 ;for spurious interrupt routine
         @ SPUR                 ;
         @ SPUR                 ;
          EVENT                 ;EVENT is address of
                                ;event interrupt handler
         @ SPUR                 ;
         @ SPUR                 ;
         @ SPUR                 ;
         @ TTIN                 ;Address of DCT for TTY in
         @ TTOUT                ;Address of DCT for TTY out
         @ SPUR                 ;Rest of table
          ...                   ;is filled with
          ...                   @ SPUR
          ...                   ;
SPUR:    @ SPURH                ;Push bit=1; SPURH = addr.
                                ; of spurious intpt handler
          0                     ;Do not change current mask
TTIN:    @ TTIH                 ;TTIH=Addr. of TTI handler
          3                     ;Mask out level 14 and 15
TTOUT:   @ TTOH                 ;TTOH=Addr. of TTO handler
          1                     ;Mask out level 15
          ...                   ;
          ...                   ;
          ...                   ;
EVENT:    ...                   ;Do processing associated
                                ;with event counter
          DSZ     LEVEL         ;
          JMP     .+1           ;
          INTEN                 ;
          JMP     @ 0           ;Return to address in loc 0
TTIH:     ...                   ;Do processing associated
          ...                   ;with TTY input
          ...                   ;
          JMP     DISMIS        ;Go to dismiss routine
TTOH:     ...                   ;Do processing associated
                                ;with TTY output
          ...                   ;
          JMP     DISMIS        ;Go to dismiss routine
```

# ERROR CHECKING AND CORRECTION

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

## Method of Operation

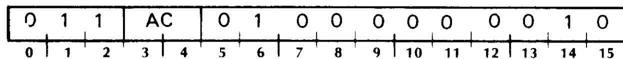
The word length of an ERCC memory is 21 bits. These 21 bits are broken into 16 data bits followed by 5 ERCC bits [COR0-COR4]. This check field is constructed by a hardware encoder from the 16 data bits and is written each time the memory location is written into. When the memory location is read, the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

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

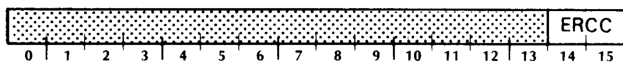
The operation of the ERCC option is governed by one I/O instruction. Two other instructions are used to interrogate the option after it has detected and corrected an error. The ERCC option has no Busy flag and no mask bit in the priority mask. The device code for the ERCC option is 2. The instructions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

## Enable ERCC

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




The ERCC option is enabled according to the setting of bits 14-15 of the specified AC. Bits 0-13 of the specified AC are ignored. The contents of the specified AC remain unchanged. The format of the specified AC is as follows:



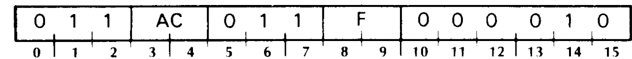

| Bits | Name | Contents |
|---|---|---|
| 0-13 | ---- | Reserved for future use. |
| 14-15 | ERCC | Control the ERCC feature as follows:<br>00 Disable checking and correction; write valid check field.<br>01 Disable checking and correction; for core memory, write check field of 11111; for semiconductior memory, do not alter the check field.<br>10 Enable checking and correction; do not interrupt on memory error.<br>11 Enable checking and correction; interrupt on memory error. |

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

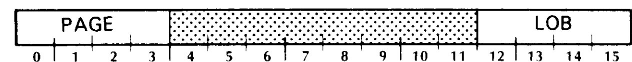
**NOTE** *When the ERCC feature detects and corrects a memory error, it sets its Done flag to 1. The Done flag will remain 1 until the ERCC feature receives a Start pulse or an I/O RESET instruction is issued. Receipt of a Start pulse will also set the fault address to 0.*

## Read Memory Fault Address

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



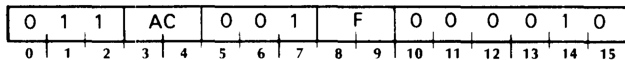

The complement of the low-order bits of the physical address of the memory location in error are placed in bits 12-15 of the specified AC. The complements of bits 1-4 of the 17-bit physical address are placed in bits 0-3 of the specified AC. The previous contents of the specified AC are lost. The format of the specified AC is as follows:



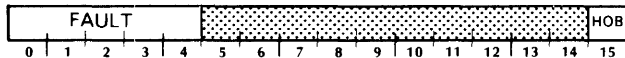

| Bits | Name | Contents |
|---|---|---|
| 0-3 | Page | Complement of bits 1-4 of the physical address of the memory location in error. |
| 4-11 | ---- | Reserved for future use. |
| 12-15 | LOB | The complement of the low-order 4 bits of the physical address of the memory location in error. |

### Read Memory Fault Code

**DIB** *[f]* *ac*, **ERCC**



A 5-bit error code is placed in bits 0-4 of the specified AC. This code tells which bit was in error and has been corrected. Bits 5-13 of the AC are set to 0. The complement of the high-order bit of the physical address of the failing location is placed in bit 15. The format of the specified AC is as follows:



| Bits | Name | Contents |
|---|---|---|
| 0-4 | Code | A 5-bit code identifying which bit was in error, as follows:<br>00000 No error<br>00001 Check bit 4<br>00010 Check bit 3<br>00011 Data bit 0<br>00100 Check bit 2<br>00101 Data bit 1<br>00110 Multiple bit error<br>00111 Data bit 3<br>01000 Check bit 1<br>01001 Data bit 4<br>01010 All 21 bits of the memory location are 1<br>01011 Data bit 6<br>01100 Data bit 7<br>01101 Data bit 8<br>01110 Data bit 9<br>01111 Multiple bit error<br>10000 Check bit 0<br>10001 Data bit 11<br>10010 Data bit 12<br>10011 Data bit 13<br>10100 Data bit 14<br>10101 All 21 bits of the memory location are 0<br>10110 Data bit 2<br>10111 Multiple bit error<br>11000 Data bit 10<br>11001 Multiple bit error<br>11010 Data bit 5<br>11011 Multiple bit error<br>11100 Data bit 15<br>11101 Multiple bit error<br>11110 Multiple bit error<br>11111 Multiple bit error |
| 5-14 | ---- | Reserved for future use. |
| 15 | HOB | Complement of the high-order bit of the physical address of the memory location in error. |

## REAL TIME CLOCK

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

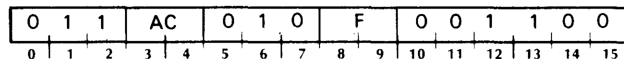
Setting Busy to 1 allows the next pulse from the clock to set Done to 1 and the RTC option requests an I/O interrupt if its priority mask bit is 0. A SELECT RTC FREQUENCY instruction to select the clock frequency only has to be given once. After each interrupt, an **NIOS RTC** instruction will set up the clock for the next interrupt.

When Busy is first set to 1, the first interrupt can come at any time up to the clock period. After the first interrupt has occurred, succeeding interrupts come at the clock frequency, provided that the program alway sets Busy to 1 before the clock period expires. After power up or I/O RESET, the clock is set to the line frequency. After power up,the line frequency pulses are available immediately, but five seconds must elapse before a steady pulse train is available from the clock for other frequencies.

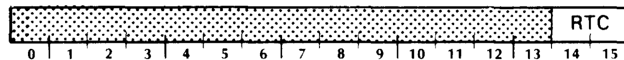
The RTC frequency is selected by the following instruction.

### Select RTC Frequency

**DOA** *[f]* *ac*, **RTC**



The clock frequency is set according to bits 14-15 of the specified AC. The contents of the specified AC remain unchanged. Bits 0-13 of the specified AC are ignored. The format of the specified AC is as follows:



| Bits | Name | Contents |
|---|---|---|
| 0-13 | ---- | Reserved for future use. |
| 14-15 | | Select the clock frequency as follows:<br>00 ac line frequency<br>01 10Hz<br>10 100Hz<br>11 1000Hz |

## 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 unless the battery back-up option is installed. Without the battery back-up option the state of the accumulators, the program counter, and the various flags in the CPU and SC memory is indeterminate. The power fail option provides a *fail-soft* capability in the event of unexpected power loss.

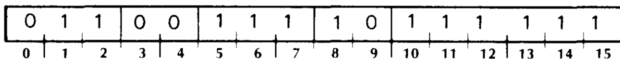
In the event of power failure, there is a delay of one to two milliseconds before the processor shuts down. The power fail option senses the loss of power, sets the Power Fail flag to 1 and requests an interrupt. The interrupt service routine can then use this delay to store the contents of the accumulators, 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 option depends upon the position of the power switch on the front panel. If the switch is in the *on* position, the CPU remains stopped after power is restored. If the switch is in the *lock* position, then 222ms after power is restored, the CPU executes the instruction contained in physical location 0, thereby transferring control to the restart procedure.

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

### Skip If Power Fail Flag Is One

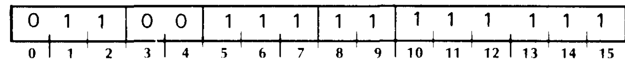
**SKPDN CPU**




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

### Skip If Power Fail Flag Is Zero

**SKPDZ CPU**




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

**NOTE** *Since the contents of semiconductor memory will be destroyed by a loss of power, (unless the battery backup option is installed) the power fail option should not attempt to restart the system, even with the power switch in the* **LOCK** *position, if the system contains semiconductor memory. This can be controlled by proper positioning of jumpers on the power fail option.*

## BATTERY BACKUP

The battery backup feature of the S/130 provides the capability to continue operation for approximately 1 minute after a power failure. During the period of battery operation, all operations of the machine are normal, with the exception of the cooling fans, which do not operate.

When power is restored, the action taken by the automatic restart portion of the battery backup feature 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 power switch is in the *lock* position, the bootstrap loader is deposited into memory locations $0\text{-}37_8$. Depending upon the position of jumpers on the CPU board, device code $33_8$ or $73_8$ (moving head disc) will be placed in AC0. When the selected device indicates ready, the CPU is started at location 2.

S 130
MONITOR
STOP
STORE
OFF
POWER
ON
LOCK
EXAM
INST
START
EXEC
CONT
DEPOSIT

# CHAPTER VI
# CONSOLE

## INTRODUCTION

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

| Light | Meaning When Lit |
|---|---|
| USER MODE | The MAP feature is translating addresses in the user mode. |
| ADDR COMPARE | Operation of the machine is suspended because the comparison requested by the ADDRESS COMPARE switch has come up true. |
| ION | The Interrupt On flag is 1. |
| CARRY | The carry bit is 1. |
| ROM ADDRESS | These 12 lights display the micro-code address of the next micro-instruction to be fetched |
| DATA | These 16 lights display what is currently on the memory bus or (if in monitor mode) in the console data register. |
| ROM ERROR | a parity error was detected in the last micro instruction addressed. The CPU halts. |
| ADDRESS | These 15 lights display what is currently on the memory address bus. |

## CONSOLE SWITCHES

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

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

## Reset-Stop

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

When this switch is pushed down, the STOP function is performed. The CPU is stopped after completing the current instruction and the next instruction. After the CPU stops, the program counter points to the next instruction to be executed. Interrupt requests made after the STOP switch is pushed and before the CPU has stopped are not honored. All outstanding data channel requests are honored before the CPU is stopped and data channel requests are honored while the machine is in the stopped state. After the CPU is stopped, the address lights display the address of the next instruction to be executed.

## Deposit-Examine

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

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

## Exam-Exam Nxt

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

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

## Inst-Micro/Inst

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

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

When this switch is pushed down, the MICRO-INSTRUCTION STEP function performed. The next microinstruction in logical sequence is performed and then the microcode processor is stopped.

The ROM address lights display the microcode address of the next microinstruction to be fetched. The address lights display the contents of the memory address bus, and the data lights display the contents of the memory bus.

## PR-Load-Exec

When this switch is pushed up, the PROGRAM LOAD function is performed. The contents of the bootstrap read-only memory are placed in memory locations 0-$37_8$ and a **JMP 0** instruction is performed.

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

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

## Start-Cont

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

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

## Dep-Dep Nxt

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

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

## Address Compare

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

### Off

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

### Monitor

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

### Stop/Store

When the rotary switch is in the STOP/STORE position, the ADDRESS COMPARE feature suspends the operation of the CPU if the CPU tries to alter the location whose address is set in the data switches. The addressed location is altered. The ADDR COMPARE light is lit to indicate that the ADDRESS COMPARE feature has suspended the operation of the machine.

### Stop/Addr

When the rotary switch is in the STOP/ADDR position, the ADDRESS COMPARE feature suspends the operation of the CPU if the CPU tries to access the location whose address is set in the data switches. The addressed location is neither read nor written. The ADDR COMPARE light is lit to indicate that the ADDRESS COMPARE feature has suspended the operation of the machine.

## Power

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

## PROGRAM LOADING

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

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

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

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

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

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

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

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

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

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

## Self Test Function

The S/130 has a system for performing a simple self-test function just before doing a program load. See the *ECLIPSE S/130 Technical Manual, DGC No. 015-000070* for details about the tests this program does.

The self-test program is automatically executed by the CPU if one of the following conditions exists:

- The console is locked (auto-restart), and the lowest memory locations are semiconductor, as determined by jumpers on CPU2;
- The PROGRAM LOAD key is depressed and switch 4 of the address/data switchess is set to 0.

If the self-test program does not produce any errors, the normal auto-restart or program load function is initiated.

If the self-test program fails, you can try to program load without using the self-test program by setting address/data switch 4 to 1 (along with the standdard program-load settings) and depressing the PROGRAM LOAD key.

If you would like to run the self-test program, but do not want to program load, set the address/data switches to 000000 and press the PROGRAM LOAD key. The self-test program will continue to run until an error occurs or address/data switch 0 is set to 1. The microprogram will then halt at ROM address 2.

**NOTE** *Setting switch 0 to 1 will stop the test only at the end of a complete run. The STOP key and the RUNNING EXAMINE key will not function while the test is executing.*

You can make the self-test program loop on errors by disabling the ROM parity detection logic as follows:

1. Start the self-test program. (Place 000000 in the address/data switches and press the PROGRAM LOAD key).
2. Turn the console power switch to LOCK.

Errors will then cause the self-test program to restart.

# 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 | | | | 43 | DIOT | 6 | Digital I/O 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 | | | |
| 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 | 10 | Cassette tape | 73 | DKP1 | 7 | Second moving head disc |
| 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 | | | |
| 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

# 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

| DECIMAL | OCTAL | HEX | KEY SYMBOL | MNEMONIC |
|---|---|---|---|---|
| 0 | 000 | 00 | | NUL |
| 1 | 001 | 01 | ↑A | SOH |
| 2 | 002 | 02 | ↑B | STX |
| 3 | 003 | 03 | ↑C | ETX |
| 4 | 004 | 04 | ↑D | EOT |
| 5 | 005 | 05 | ↑E | ENQ |
| 6 | 006 | 06 | ↑F | ACK |
| 7 | 007 | 07 | ↑G | BEL |
| 8 | 010 | 08 | ↑H | BS (BACKSPACE) |
| 9 | 011 | 09 | ↑I | TAB |
| 10 | 012 | 0A | ↑J | NEW LINE |
| 11 | 013 | 0B | ↑K | VT (VERT. TAB) |
| 12 | 014 | 0C | ↑L | FORM FEED |
| 13 | 015 | 0D | ↑M | CARRIAGE RETURN |
| 14 | 016 | 0E | ↑N | SO |
| 15 | 017 | 0F | ↑O | SI |
| 16 | 020 | 10 | ↑P | DLE |
| 17 | 021 | 11 | ↑Q | DC1 |
| 18 | 022 | 12 | ↑R | DC2 |
| 19 | 023 | 13 | ↑S | DC3 |
| 20 | 024 | 14 | ↑T | DC4 |
| 21 | 025 | 15 | ↑U | NAK |
| 22 | 026 | 16 | ↑V | SYN |
| 23 | 027 | 17 | ↑W | ETB |
| 24 | 030 | 18 | ↑X | CAN |
| 25 | 031 | 19 | ↑Y | EM |
| 26 | 032 | 1A | ↑Z | SUB |
| 27 | 033 | 1B | ESC | ESCAPE |
| 28 | 034 | 1C | ↑\ | FS |
| 29 | 035 | 1D | ↑] | GS |
| 30 | 036 | 1E | ↑↑ | RS |
| 31 | 037 | 1F | ↑← | US |

| DECIMAL | OCTAL | HEX | KEY SYMBOL |
|---|---|---|---|
| 32 | 040 | 20 | SPACE |
| 33 | 041 | 21 | ! |
| 34 | 042 | 22 | " (QUOTE) |
| 35 | 043 | 23 | # |
| 36 | 044 | 24 | $ |
| 37 | 045 | 25 | % |
| 38 | 046 | 26 | & |
| 39 | 047 | 27 | ' (APOS) |
| 40 | 050 | 28 | ( |
| 41 | 051 | 29 | ) |
| 42 | 052 | 2A | * |
| 43 | 053 | 2B | + |
| 44 | 054 | 2C | , (COMMA) |
| 45 | 055 | 2D | - |
| 46 | 056 | 2E | . (PERIOD) |
| 47 | 057 | 2F | / |
| 48 | 060 | 30 | 0 |
| 49 | 061 | 31 | 1 |
| 50 | 062 | 32 | 2 |
| 51 | 063 | 33 | 3 |
| 52 | 064 | 34 | 4 |
| 53 | 065 | 35 | 5 |
| 54 | 066 | 36 | 6 |
| 55 | 067 | 37 | 7 |
| 56 | 070 | 38 | 8 |
| 57 | 071 | 39 | 9 |
| 58 | 072 | 3A | : |
| 59 | 073 | 3B | ; |
| 60 | 074 | 3C | < |
| 61 | 075 | 3D | = |
| 62 | 076 | 3E | < |
| 63 | 077 | 3F | ? |
| 64 | 100 | 40 | @ |

| DECIMAL | OCTAL | HEX | KEY SYMBOL |
|---|---|---|---|
| 65 | 101 | 41 | A |
| 66 | 102 | 42 | B |
| 67 | 103 | 43 | C |
| 68 | 104 | 44 | D |
| 69 | 105 | 45 | E |
| 70 | 106 | 46 | F |
| 71 | 107 | 47 | G |
| 72 | 110 | 48 | H |
| 73 | 111 | 49 | I |
| 74 | 112 | 4A | J |
| 75 | 113 | 4B | K |
| 76 | 114 | 4C | L |
| 77 | 115 | 4D | M |
| 78 | 116 | 4E | N |
| 79 | 117 | 4F | O |
| 80 | 120 | 50 | P |
| 81 | 121 | 51 | Q |
| 82 | 122 | 52 | R |
| 83 | 123 | 53 | S |
| 84 | 124 | 54 | T |
| 85 | 125 | 55 | U |
| 86 | 126 | 56 | V |
| 87 | 127 | 57 | W |
| 88 | 130 | 58 | X |
| 89 | 131 | 59 | Y |
| 90 | 132 | 5A | Z |
| 91 | 133 | 5B | [ |
| 92 | 134 | 5C | \ |
| 93 | 135 | 5D | ] |
| 94 | 136 | 5E | ↑ OR ∧ |
| 95 | 137 | 5F | ← OR — |
| 96 | 140 | 60 | ` (GRAVE) |

| DECIMAL | OCTAL | HEX | KEY SYMBOL |
|---|---|---|---|
| 97 | 141 | 61 | a |
| 98 | 142 | 62 | b |
| 99 | 143 | 63 | c |
| 100 | 144 | 64 | d |
| 101 | 145 | 65 | e |
| 102 | 146 | 66 | f |
| 103 | 147 | 67 | g |
| 104 | 150 | 68 | h |
| 105 | 151 | 69 | i |
| 106 | 152 | 6A | j |
| 107 | 153 | 6B | k |
| 108 | 154 | 6C | l |
| 109 | 155 | 6D | m |
| 110 | 156 | 6E | n |
| 111 | 157 | 6F | o |
| 112 | 160 | 70 | p |
| 113 | 161 | 71 | q |
| 114 | 162 | 72 | r |
| 115 | 163 | 73 | s |
| 116 | 164 | 74 | t |
| 117 | 165 | 75 | u |
| 118 | 166 | 76 | v |
| 119 | 167 | 77 | w |
| 120 | 170 | 78 | x |
| 121 | 171 | 79 | y |
| 122 | 172 | 7A | z |
| 123 | 173 | 7B | { |
| 124 | 174 | 7C | \| |
| 125 | 175 | 7D | } |
| 126 | 176 | 7E | ~ (TILDE) |
| 127 | 177 | 7F | DEL (RUBOUT) |

# 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):

$(1x2^6) + (0x2^5) + (1x2^4) + (1x2^3) +$
$(0x2^2) + (0x2^1) + (1x2^0)$

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 3 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

$1111 = 2^3 + 2^2 + 2^1 + 2^0$
$= (8 + 4 + 2 + 1) = 15_{10} = F_{16}$

The number in the example above is then:

101 1010

or

5 A = $5A_{16}$

Conversion to decimal is not quite so simple. Tables in Appendix B of this manual can be used to convert between octal and decimal or between hexadecimal and decimal. Using them, we find that the above number is $90_{10}$.

The table below shows the correspondence among the four numbering systems mentioned above for a 4-bit binary number:

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

In this manual, numbers without a subscript will be assumed to be in base 10, or decimal, representation. All other representations will be explicitly noted in a subscript, thus:

$$64_{10} = 40_{16} = 100_8$$

$$63_{10} = 3F_{16} = 77_8$$

# APPENDIX E
# COMPATIBILITY WITH NOVA LINE COMPUTERS

The ECLIPSE S/130 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
# INSTRUCTION EXECUTION TIMES

The following table gives typical execution times for representative instructions in the S/130 instruction set. These times assume a system with semiconductor memory and the floating point instruction set option. All times are in microseconds.

| Instruction | Execution Time |
|---|---|
| ADD | 0.6 |
| BLM | 2.8 + 1.2/word |
| CMV | 1.8 + 2.9/byte |
| DIV | 4.3 |
| DIVS | 5.85 |
| DIVX | 5.8 |
| FAS | 7.25 |
| FMD | 58.6 |
| FMS | 13.6 |
| LDA | 1.2 |
| MUL | 3.2 |
| MULS | 4.0 |
| STA | 1.2 |
| SUB | 0.6 |
| VCT | |
| Short | 2.6 |
| Long | 20.4 |

# APPENDIX G
# INSTRUCTION USE EXAMPLES

On the following pages are examples of how the instruction set of the ECLIPSE S/130 may be used to perform some common functions.

1. Clear an AC and the carry bit.

```
SUBO   AC,AC
```

2. Clear an AC and preserve the carry bit.

```
SUBC   AC,AC
```

3. Generate the indicated constants.

```
SUBZL      AC,AC      ;Generate +1
ADC        AC,AC      ;Generate -1
ADCZL      AC,AC      ;Generate -2
```

4. Let ACN be any accumulator whose contents are zero; generate the indicated constants in ACN.

```
INCZL      ACN,ACN      ;Generate +2
INCOL      ACN,ACN      ;Generate +3
INCS       ACN,ACN      ;Generate +400 8
```

5. Check if both bytes in an accumulator are equal.

```
MOVS       ACS,ACD
SUB        ACS,ACD,SZR
JMP        ---           ;Not equal
---        ---           ;Equal
```

6. Check if two accumulators are both zero.

```
MOVS       ACS,ACS,SNR
SUB        ACS,ACD,SZR
JMP        ---           ;Not equal
---        ---           ;Equal
```

7. Check an ASCII character to make sure it is a decimal digit. The character is in ACS and is not destroyed by the test. Accumulators ACx and ACy are destroyed.

```
LDA        ACx,C60           ;ASCII zero
LDA        ACy,C71           ;ASCII nine
ADCZ #     ACy,ACS,SNC       ;Skips if (ACS>9)
ADCZ #     ACS,ACx,SZC       ;Skips if (ACS≥0)
JMP        ---               ;Not digit
---        ---               ;Digit
C60:       60                ;ASCII 0
C71:       71                ;ASCII 9
```

8. Test an accumulator for zero.

```
MOV        AC,AC,SZR
JMP        ---           ;Not zero
---        ---           ;Zero
```

9. Test an accumulator for -1.

```
COM #      AC,AC,SZR
JMP        ---           ;Not -1
---        ---           ;-1
```

10. Test an accumulator for 2 or greater.

```
MOVZR #    AC,AC,SNR
JMP        ---          ;Less than 2
---        ---          ;2 or greater
```

11. Assume that it is known that AC contains 0, 1, 2, or 3; find out which value.

```
MOVZR #    AC,AC,SEZ
JMP        THREE        ;Was 3
MOV        AC,AC,SNR
JMP        ZERO         ;Was 0
MOVZR #    AC,AC,SZR
JMP        TWO          ;Was 2
---        ---          ;Was 1
```

12. Perform the following unsigned integer comparisons.

```
SUB #      ACS,ACD,SZR     ;Skip if (ACS) = (ACD)
SUB #      ACS,ACD,SNR     ;Skip if (ACS) ≠ (ACD)
ADCZ #     ACS,ACD,SNC     ;Skip if (ACS) < (ACD)
SUBZ #     ACS,ACD,SNC     ;Skip if (ACS) ≤ (ACD)
SUBZ #     ACS,ACD,SZC     ;Skip if (ACS) > (ACD)
ADCZ #     ACS,ACD,SZC     ;Skip if (ACS) ≥ (ACD)
```

13. Multiply an AC by the indicated value.

```
MOV        ACx,ACx      ;Multiply by 1

MOVZL      ACx,ACx      ;Multiply by 2

MOVZL      ACx,ACy      ;Multiply by 3
ADD        ACy,ACx

ADDZL      ACx,ACx      ;Multiply by 4

MOV        ACx,ACy      ;Multiply by 5
ADDZL      ACx,ACx
ADD        ACy,ACx

MOVZL      ACx,ACy      ;Multiply by 6
ADDZL      ACy,ACx

ADDZL      ACx,ACx      ;Multiply by 8
MOVZL      ACx,ACx
```

Multiplication by other factors of 2 can be achieved with the LOGICAL SHIFT instruction; multiplication by factors of 16 can be accomplished with the HEX SHIFT LEFT instruction.

# INSTRUCTION INDEX


ADC (Add Complement) **III-6**, III-8*
ADD (Add) **III-3**, III-8, III-27, VI-4
ADDI (Extended Add Immediate) **III-5**
ADI (Add Immediate) **III-5**, III-30, III-31
ANC (AND With Complemented Source) **III-10**
AND (Logical AND) **III-9**, VI-4
ANDI (AND Immediate) **III-9**

BAM (Block Add And Move) **III-19**, III-20
BLM (Block Move) **III-19**
BTO (Set Bit To One) **III-16**
BTZ (Set Bit To Zero) **III-16**

CLM (Compare To Limits) **III-14**
CMP (Character Compare) **IV-20**
CMT (Character Move Until True) **IV-22**
CMV (Character Move) **IV-19**, IV-23
COB (Count Bits) **III-18**
COM (Complement) **III-9**, V-13, VI-4
CTR (Character Translate) **IV-21**

DAD (Decimal Add) **III-4**
DHXL (Double Hex Shift Left) **III-11**
DHXR (Double Hex Shift Right) **III-12**
DIA (Data In A) **V-3**, V-4
DIA *ac*, CPU (Read Switches) **V-9**, VI-4
DIA *ac*, ERCC (Read Memory Fault Address) **V-14**
DIA *ac*, MAP (Read MAP Status) **IV-6**
DIB (Data In B) **V-3**
DIB *ac*, CPU (Interrupt Acknowledge) V-5, V-6, **V-8**
DIB *ac*, ERCC (Read Memory Fault Code) **V-15**
DIC (Data In C) **V-3**
DIC *ac*, CPU (I/O Reset) IV-4, IV-16, **V-9**, VI-2, VI-4
DIC *ac*, MAP (Page Check) **IV-7**
DIV (Unsigned Divide) **III-7**
DIVS (Signed Divide) **III-7**
DIVX (Sign Extend And Divide) **III-7**
DLSH (Double Logical Shift) **III-11**, III-16
DOA (Data Out A) **V-3**, V-4
DOA *ac*, ERCC (Enable ERCC) **V-14**
DOA *ac*, MAP (Load MAP Status) **IV-6**
DOA *ac*, RTC (Select RTC Frequency) **V-13**
DOB (Data Out B) **V-3**
DOB *ac*, CPU (Mask Out) **V-8**, V-13
DOB *ac*, MAP (Map Supervisor Page 31) **IV-8**
DOC (Data Out C) **V-4**
DOC *ac*, CPU (Halt) **V-9**
DOC *ac*, MAP (Initiate Page Check) **IV-7**
DSB (Decimal Subtract) **III-4**
DSPA (Dispatch) **III-14**, III-15
DSZ (Decrement And Skip If Zero) **III-13**, III-16, V-13


* Primary references are in **bold type.** Others are in light type.

EDSZ (Extended Decrement And Skip If Zero) **III-13**
EISZ (Extended Increment And Skip If Zero) **III-13**
EJMP (Extended Jump) **III-28**
EJSR (Extended Jump To Subroutine) **III-28**
ELDA (Extended Load Accumulator) **III-3**
ELDB (Extended Load Byte) IV-23, **IV-24**
ELEF (Extended Load Effective Address) **III-9**
ESTA (Extended Store Accumulator) **III-3**
ESTB (Extended Store Byte) **IV-24**

FAB (Absolute Value) **IV-14**
FAD (Add Double (FPAC to FPAC) **IV-12**
FAMD (Add Double (memory to FPAC) **IV-12**
FAMS (Add Single (memory to FPAC) **IV-12**
FAS (Add Single (FPAC to FPAC) **IV-12**
FCLE (Clear Errors) **IV-16**
FCMP (Compare Floating Point) **IV-16**
FDD (Divide Double (FPAC by FPAC) **IV-14**
FDMD (Divide Double (FPAC by memory) **IV-14**
FDMS (Divide Single (FPAC by memory) **IV-13**
FDS (Divide Single (FPAC by FPAC) **IV-13**
FEXP (Load Exponent) **IV-15**
FFAS (Fix To AC) IV-9, **IV-11**
FFMD (Fix To Memory) IV-9, **IV-11**
FHLV (Halve) **IV-15**
FINT (Integerize) **IV-15**
FLAS (Float From AC) **IV-11**
FLDD (Load Floating Point Double) **IV-10**
FLDS (Load Floating Point Single) **IV-10**
FLMD (Float From Memory) **IV-11**
FLST (Load Floating Point Status) **IV-16**
FMD (Multiply Double (FPAC by FPAC) **IV-13**
FMMD (Multiply Double (FPAC by memory) **IV-13**
FMMS (Multiply Single (FPAC by memory) **IV-13**
FMOV (Floating Point Move) **IV-11**
FMS (Multiply Single (FPAC by FPAC) **IV-13**
FNEG (Negate) **IV-14**
FNOM (Normalize) **IV-14**
FNS (No Skip) **IV-18**
FPOP (Pop Floating Point State) IV-10, **IV-17**
FPSH (Push Floating Point State) IV-10, **IV-17**
FRH (Read High Word) **IV-14**
FSA (Skip Always) **IV-18**
FSCAL (Scale) IV-9, **IV-15**
FSD (Subtract Double (FPAC from FPAC) **IV-12**
FSEQ (Skip On Zero) **IV-18**
FSGE (Skip On Greater Than Or Equal To Zero) **IV-18**
FSGT (Skip On Greater Than Zero) **IV-18**
FSLE (Skip On Less Than Or Equal To Zero) **IV-18**
FSLT (Skip On Less Than Zero) **IV-18**
FSMD (Subtract Double (memory from FPAC) **IV-13**
FSMS (Subtract Single (memory from FPAC) **IV-12**
FSND (Skip On No Zero Divide) **IV-19**
FSNE (Skip On Non-Zero) **IV-18**
FSNER (Skip On No Error) **IV-19**
FSNM (Skip On No Mantissa Overflow) **IV-18**
FSNO (Skip On No Overflow) **IV-19**
FSNOD (Skip On No Overflow And No Zero Divide) **IV-19**
FSNU (Skip On No Underflow) **IV-18**
FSNUD (Skip On No Underflow And No Zero Divide) **IV-19**
FSNUO (Skip On No Underflow And No Overflow) **IV-19**
FSS (Subtract Single (FPAC from FPAC)) **IV-12**
FSST (Store Floating Point Status) **IV-16**
FSTD (Store Floating Point Double) **IV-10**
FSTS (Store Floating Point Single) **IV-10**
FTD (Trap Disable) **IV-16**
FTE (Trap Enable) **IV-16**

HALT (Halt) **V-9**
HALTA (Halt) **V-9**
HLV (Halve) **III-7**
HXL (Hex Shift Left) **III-11**, III-20
HXR (Hex Shift Right) **III-11**, III-16

INC (Increment) **III-6**, III-8, III-18, III-20, VI-4
INTA (Interrupt Acknowledge) V-5, V-6, **V-8**
INTDS (Interrupt Disable) **V-8**
INTEN (Interrupt Enable) V-5, **V-8**, V-13
IOR (Inclusive OR) **III-10**, III-16
IORI (Inclusive OR Immediate) **III-10**
IORST (I/O Reset) IV-4, IV-16, **V-9**, VI-2, VI-4
ISZ (Increment And Skip If Zero) **III-13**, V-13, VI-4

JMP (Jump) III-15, III-16, III-18, III-20, **III-28**, III-29, III-30, III-31, IV-23, V-4, V-13, VI-4
JSR (Jump To Subroutine) III-18, III-27, **III-28**, III-30, VI-4

LCSF (Load Control Store Formatted) **IV-25**
LDA (Load Accumulator) **III-3**, III-16, III-18, III-20, III-27, III-29, III-30, III-31, IV-23, V-13, VI-2
LDB (Load Byte) **III-15**
LEF (Load Effective Address) **III-9**, III-18, IV-4, IV-23, VI-2
LOB (Locate Lead Bit) **III-17**
LMP (Load Map) **IV-5**
LRB (Locate And Reset Lead Bit) **III-18**
LSH (Logical Shift) **III-10**

MOV (Move) **III-6**, III-8, III-16, III-20, III-31, IV-23, VI-4
MSKO (Mask Out) **V-8**, V-13
MSP (Modify Stack Pointer) **III-27**
MUL (Unsigned Multiply) **III-6**, III-31
MULS (Signed Multiply) **III-7**

NEG (Negate) **III-5**, IV-23
NIO (No I/O Transfer) **V-3**
NIOC CPU (Interrupt Disable) **V-8**
NIOP MAP (Map Single Cycle) **IV-8**
NIOS CPU (Interrupt Enable) V-5, **V-8**, V-13

POP (Pop Multiple Accumulators) **III-25**, III-31, V-13
POPB (Pop Block) III-18, **III-29**, V-13
POPJ (Pop PC And Jump) **III-29**, III-31
PSH (Push Multiple Accumulators) **III-25**, III-31
PSHJ (Push Jump) **III-29**, III-31
PSHR (Push Return Address) **III-26**

READS (Read Switches) **V-9**, VI-4
RSTR (Restore) **III-28**, V-13
RTN (Return) **III-27**, III-31

SAVE (Save) **III-26**, III-31
SBI (Subtract Immediate) **III-5**, III-16, III-31, V-13
SGE (Skip If ACS Greater Than Or Equivalent To ACD) **III-13**
SGT (Skip If ACS Greater Than ACD) **III-13**
SKP (I/O Skip) **V-4**, V-5
SKP CPU (CPU Skip) **V-8**
SKPDN CPU (Skip If Power Fail Flag Is One) **V-16**
SKPDZ CPU (Skip If Power Fail Flag Is Zero) **V-16**
SNB (Skip On Non-Zero Bit) **III-17**, III-18
STA (Store Accumulator) **III-3**, III-16, III-20, III-27, III-30, III-31, V-13, VI-4
STB (Store Byte) **III-15**, III-16, III-18
SUB (Subtract) **III-3**, III-8, III-20, III-31, VI-4
SVC (same as SYC 0,0) **III-29**
SYC (System Call) **III-29**
SYL (same as SYC 1,1) **III-29**
SZB (Skip On Zero Bit) **III-17**
SZBO (Skip On Zero Bit And Set To One) **III-17**

VCT (Vector On Interrupting Device Code) II-8, III-27, V-5, V-6, **V-10ff**

XCH (Exchange Accumulators) **III-6**
XCT (Execute) **III-29**, VI-2
XOP (Extended Operation) **III-32**
XOP1 (Extended Operation) III-32, **IV-27**
XOR (Exclusive OR) **III-10**
XORI (Exclusive OR Immediate) **III-10**

# 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, NOVA Line Computers,
DGC No. 015-000023.

Programmer's Reference, ECLIPSE Line Computers,
DGC No.015-000024.

Programmer's Reference, ECLIPSE C-Series Computers,
DGC No. 015-000047.

Programmer's Reference, Data Control Unit,
DGC No. 015-000060.

Programmer's Reference, DGDAC and Process Controls,
DGC No. 015-000063.

Programmer's Reference, ECLIPSE S/130,
DGC No. 015-000068.

Programmer's Reference, S/130 Microprogramming WCS Feature,
DGC No. 015-000069.

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, ECLIPSE S/130,
DGC No. 015-000070.

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.

## SALES AND SERVICE OFFICES

**Alabama:** Birmingham
**Arizona:** Phoenix, Tucson
**Arkansas:** Little Rock
**California:** El Segundo, Fresno, Palo Alto, Sacramento, San Diego, San Francisco, Santa Ana, Santa Barbara, Van Nuys
**Colorado:** Englewood
**Connecticut:** North Branford
**Florida:** Ft. Lauderdale, Orlando, Tampa
**Georgia:** Norcross
**Idaho:** Boise
**Illinois:** Peoria, Schaumburg
**Indiana:** Indianapolis
**Kentucky:** Louisville
**Louisiana:** Baton Rouge
**Maryland:** Baltimore
**Massachusetts:** Springfield, Wellesley, Worcester
**Michigan:** Southfield
**Minnesota:** Richfield
**Missouri:** Kansas City, St. Louis
**Nevada:** Las Vegas
**New Hampshire:** Nashua
**New Jersey:** Cherry Hill, Wayne
**New Mexico:** Albuquerque
**New York:** Buffalo, Latham, Melville, Newfield, New York, Rochester, Syracuse, White Plains
**North Carolina:** Charlotte, Greensboro
**Ohio:** Columbus, Dayton, Brooklyn Heights
**Oklahoma:** Oklahoma City, Tulsa
**Oregon:** Portland
**Pennsylvania:** Blue Bell, Carnegie
**Rhode Island:** Rumford
**South Carolina:** Columbia
**Tennessee:** Knoxville, Memphis
**Texas:** Austin, Dallas, El Paso, Ft. Worth, Houston
**Utah:** Salt Lake City
**Virginia:** McLean, Norfolk, Richmond, Salem
**Washington:** Kirkland
**West Virginia:** Charleston
**Wisconsin:** West Allis

**Australia:** Melbourne, Victoria
**France:** Le Plessis Robinson
**Italy:** Milan, Padua, Rome
**The Netherlands:** Rijswijk
**New Zealand:** Auckland, Wellington
**Sweden:** Gothenburg, Malmoe, Stockholm
**Switzerland:** Lausanne, Zurich
**United Kingdom:** Birmingham, Dublin, Glasgow, London, Manchester
**West Germany:** Filderstadt, Frankfurt, Hamburg, Munich, Ratingen, Rodelheim

## MANUFACTURER'S REPRESENTATIVES & DISTRIBUTORS

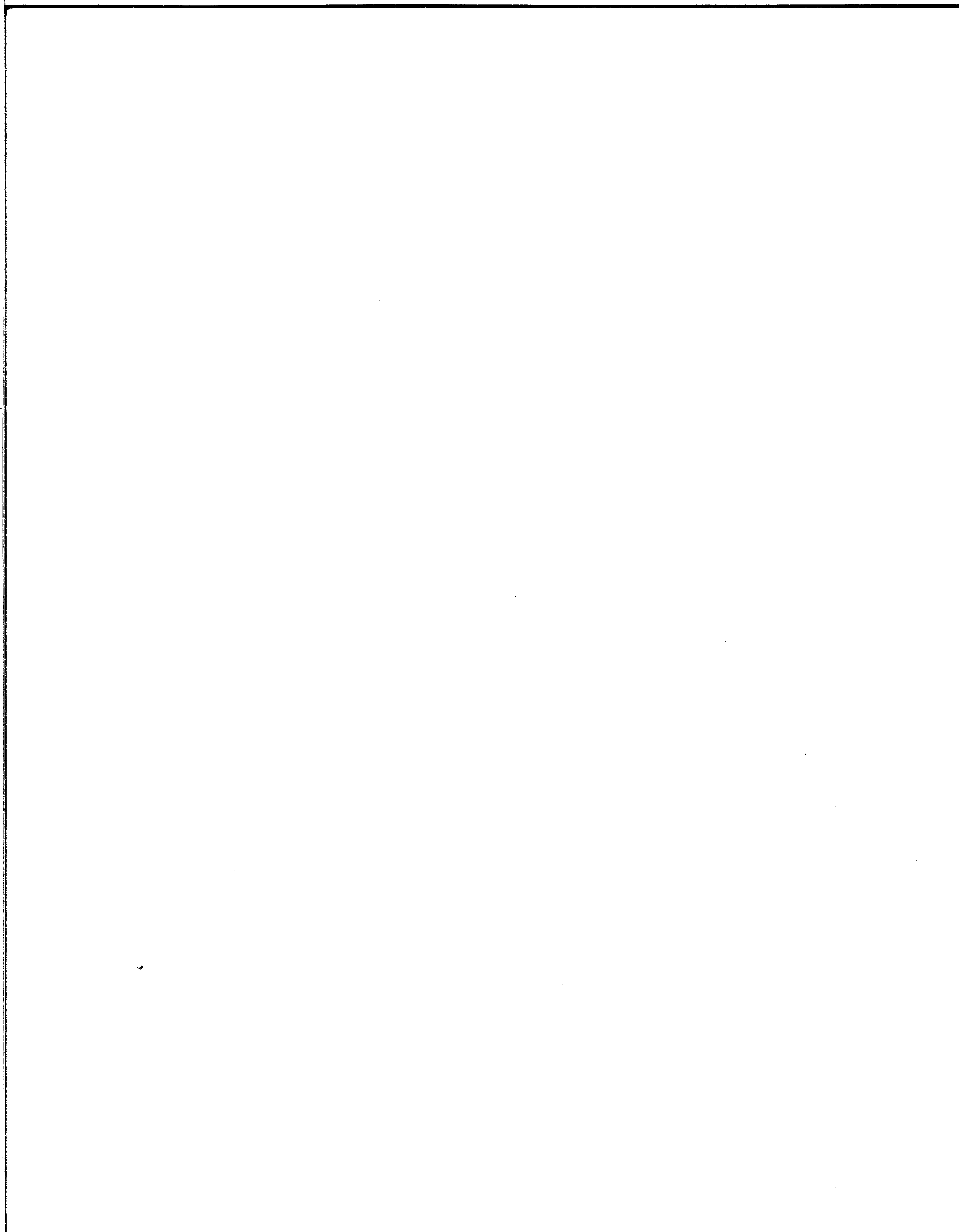
**Argentina:** Buenos Aires
**Costa Rica:** San Jose
**Ecuador:** Quito
**Egypt:** Cairo
**Finland:** Helsinki
**Greece:** Athens
**Hong Kong:** Hong Kong
**India:** Bombay
**Indonesia:** Jakarta
**Iran:** Tehran
**Israel:** Givatayim
**Japan:** Tokyo
**Jordan:** Amman
**Korea:** Seoul
**Kuwait:** Kuwait
**Lebanon:** Beirut
**Malaysia:** Kuala Lumpur
**Mexico:** Mexico City
**Nicaragua:** Managua
**Nigeria:** Lagos, Ibadan
**Norway:** Oslo
**Peru:** Lima
**Philippine Islands:** Manila
**Puerto Rico:** Hato Rey
**Saudi Arabia:** Riyadh
**Singapore:** Singapore:
**South Africa:** Johannesburg, Pretoria
**Spain:** Barcelona, Bilbao, Madrid, San Sebastian, Valencia
**Taiwan:** Taipei
**Thailand:** Bangkok
**Uruguay:** Montevideo
**Venezuela:** Maracaibo

## ADMINISTRATION, MANUFACTURING RESEARCH AND DEVELOPMENT

**Massachusetts:** Cambridge, Framingham, Westboro, Southboro
**Maine:** Westbrook
**New Hampshire:** Portsmouth
**California:** Anaheim, Sunnyvale
**North Carolina:** Research Triangle Park, Johnston County

**Hong Kong:** Kowloon, Tai Po
**Thailand:** Bangkok

*DG-04976*



Data General Corporation, Westboro, Massachusetts 01581

014-000621-01 Copyright ©Data General Corporation, 1979