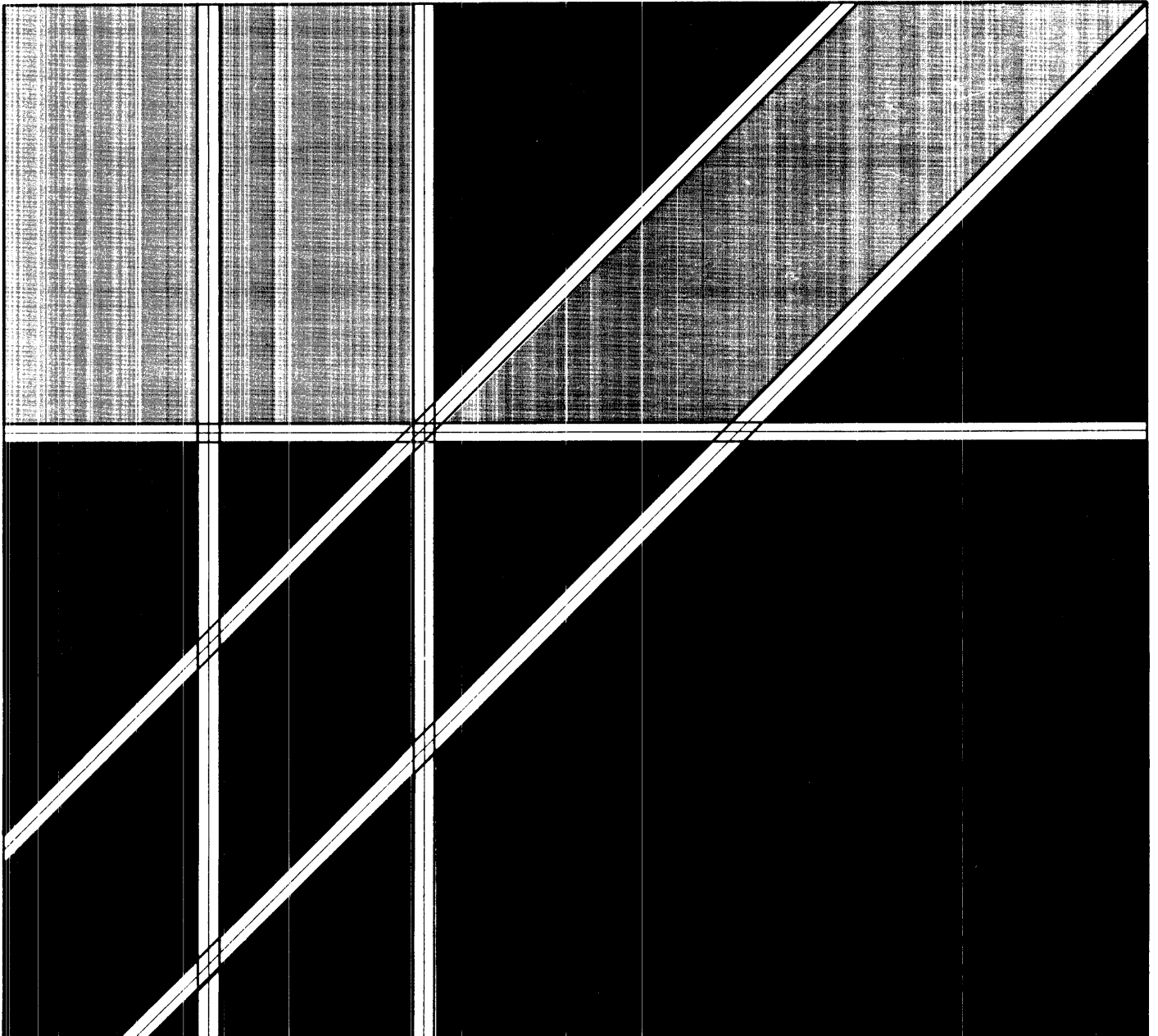


Four-Phase Systems



System IV/70

Computer Reference Manual



Four-Phase Systems, Inc.
10420 North Tantau Avenue
Cupertino, California 95014

Document Number SIV/70-11-1C

Stock Number 33C

Issue A: 1 November 1970

Issue B: 1 April 1971

Issue C: 1 October 1972

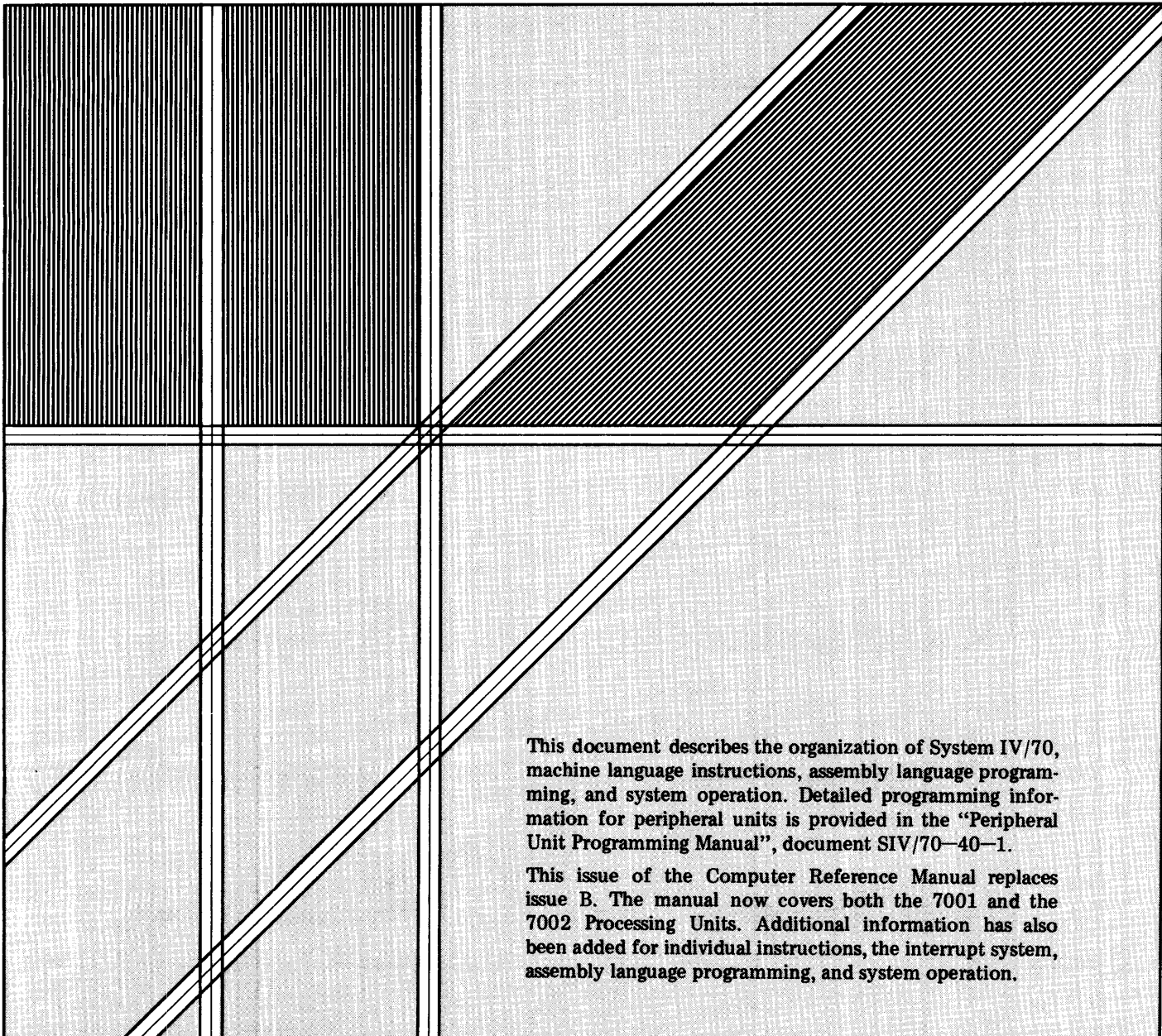
Specifications subject to change.

Copyright 1972, 1971, 1970 Four-Phase Systems, Inc.
All rights reserved.

Price: \$4.50

System IV/70

Computer Reference Manual



This document describes the organization of System IV/70, machine language instructions, assembly language programming, and system operation. Detailed programming information for peripheral units is provided in the "Peripheral Unit Programming Manual", document SIV/70-40-1.

This issue of the Computer Reference Manual replaces issue B. The manual now covers both the 7001 and the 7002 Processing Units. Additional information has also been added for individual instructions, the interrupt system, assembly language programming, and system operation.

LIST OF EFFECTIVE PAGES

THIS PUBLICATION CONTAINS 110 PAGES
CONSISTING OF THE FOLLOWING:

<i>Page Number</i>	<i>Issue</i>	<i>Page Number</i>	<i>Issue</i>
Front Cover	---		
Inside Front Cover (Blank)	---		
Title Page	1 Oct 72		
A	1 Oct 72		
i thru iv	1 Oct 72		
1-1	1 Oct 72		
1-2 (Blank)	1 Oct 72		
2-1 thru 2-2	1 Oct 72		
3-1 thru 3-10	1 Oct 72		
4-1 thru 4-24	1 Oct 72		
5-1 thru 5-17	1 Oct 72		
5-18 (Blank)	1 Oct 72		
6-1 thru 6-5	1 Oct 72		
6-6 (Blank)	1 Oct 72		
7-1 thru 7-3	1 Oct 72		
7-4 (Blank)	1 Oct 72		
8-1 thru 8-9	1 Oct 72		
8-10 (Blank)	1 Oct 72		
9-1 thru 9-7	1 Oct 72		
9-8 (Blank)	1 Oct 72		
A-1 thru A-4	1 Oct 72		
B-1	1 Oct 72		
B-2 (Blank)	1 Oct 72		
C-1 thru C-5	1 Oct 72		
C-6 (Blank)	1 Oct 72		
D-1	1 Oct 72		
D-2 (Blank)	1 Oct 72		
User's Comments	---		
Reply Card	---		
Inside Back Cover (Blank)	---		
Back Cover	---		

* The asterisk indicates pages changed, added, or deleted by the current change. The portion of the text affected by the current change (or the portion of the text changed for this issue) is indicated by a vertical line in the outer margins of the page.

Contents

1 Introduction	
2 System Organization	
Introduction	2-1
Control Logic	2-1
Arithmetic Logic	2-1
Main Memory	2-1
Input/Output System	2-2
Priority Interrupt System	2-2
I/O Logic	2-2
I/O Controllers	2-2
Peripheral Units	2-2
Video Output Circuits	2-2
3 Machine Language Programming	
Introduction	3-1
Formats	3-1
Data Formats	3-2
Instruction Format	3-3
Instructions	3-3
Memory Reference Instructions	3-3
Indexing	3-3
Indirect Addressing	3-4
Non-Memory Reference Instructions	3-4
Source and Destination Field	3-4
Byte Control Field	3-5
Count or Shift Count Field	3-5
Decimal Option Instructions	3-5
Hardware Organization	3-5
Main Storage	3-5
Central Processing Unit	3-5
Control Logic	3-5
B3 Interface	3-5
Arithmetic Logic	3-7
Working Registers	3-7
Condition Codes	3-7
Arithmetic Trap	3-8
Programming for the Video/Keyboard	3-8
Video Display	3-8
Keyboard	3-9
Instruction Descriptions	3-9
4 Word Oriented Instructions	
Load/Store Instructions	4-1
Fixed-Point Arithmetic Instructions	4-3
Floating Point Instructions	4-6
Comparison Instructions	4-8
Shift Instructions	4-9
Branch and Skip Instructions	4-12
Unconditional Branch Instructions	4-12
Conditional Branch Instructions	4-14
Branch and Count Instructions	4-15
Skip Instructions	4-16
Register-to-Register Instructions	4-17
Logical Instructions	4-21
Control Instructions	4-23
5 String Manipulation Instructions	
Word and Character Manipulation Instructions	5-1
Word Move Instructions	5-1
Character Manipulation Instructions	5-2
Character Translate Instruction	5-8
List Processing Instructions	5-10
Whole Word Stack Instructions	5-10
Character List Processing Instructions	5-11
Decimal Option Instructions	5-15
6 Input/Output System and Instructions	
Peripheral Unit I/O	6-1
Organization	6-1
Device Address	6-2
Peripheral Units	6-2
I/O Instructions	6-2
Execution of I/O Instructions	6-4
External Command and External Sense I/O	6-5
Console Keys Input	6-5
7 Interrupt System and Instructions	
Priority Interrupt Levels	7-1
Interrupt Processing	7-1
Normal Interrupt Processing	7-1
Single Instruction Processing	7-2
Indirect Interrupt Processing	7-2
Interrupt Controls	7-2
Non-Interruptable Instructions	7-2
Interrupt Instructions	7-3
8 Assembly Language Programming	
General	8-1
Assembler Programming	8-1
Programs and Routines	8-1
Program Elements	8-1
Statements	8-1
Symbols	8-1
Expressions	8-2
Assembler Language Coding	8-2
Absolute and Relocatable Code	8-4
Sequence of Events	8-5
Assembler Instructions	8-5
General	8-5
Data Control Directives	8-6
Assembler Control Directives	8-7
Error Conditions	8-9
9 System Operation	
Introduction	9-1
Manual Data Display and Entry	9-1
Register Data Display	9-1

Memory Data Display 9-1
 Register Data and Condition Code Entry. 9-1
 Memory Data Entry 9-1
Program Execution. 9-2
 Automatic Execution 9-2
 Manual Execution 9-2
 Repeated Instruction Execution 9-2
Initializing the System 9-4
 Turning Power On 9-4
 Bootstrap Loading for Systems With a
 BOOT Switch 9-4
 Bootstrap Loading for Systems Without a
 BOOT Switch 9-4
Miscellaneous Procedures. 9-5

Using the Interrupt Switch. 9-5
 Halting and Clearing Errors 9-5
 Automatically Entering a Word into a Register. 9-6
Tape Drive Operating Procedures 9-6
 Loading Tape 9-6
 Unloading Tape 9-6
 Controls and Indicators 9-7

A System IV/70 Character Set

B Assembler Directives

C Machine Instructions

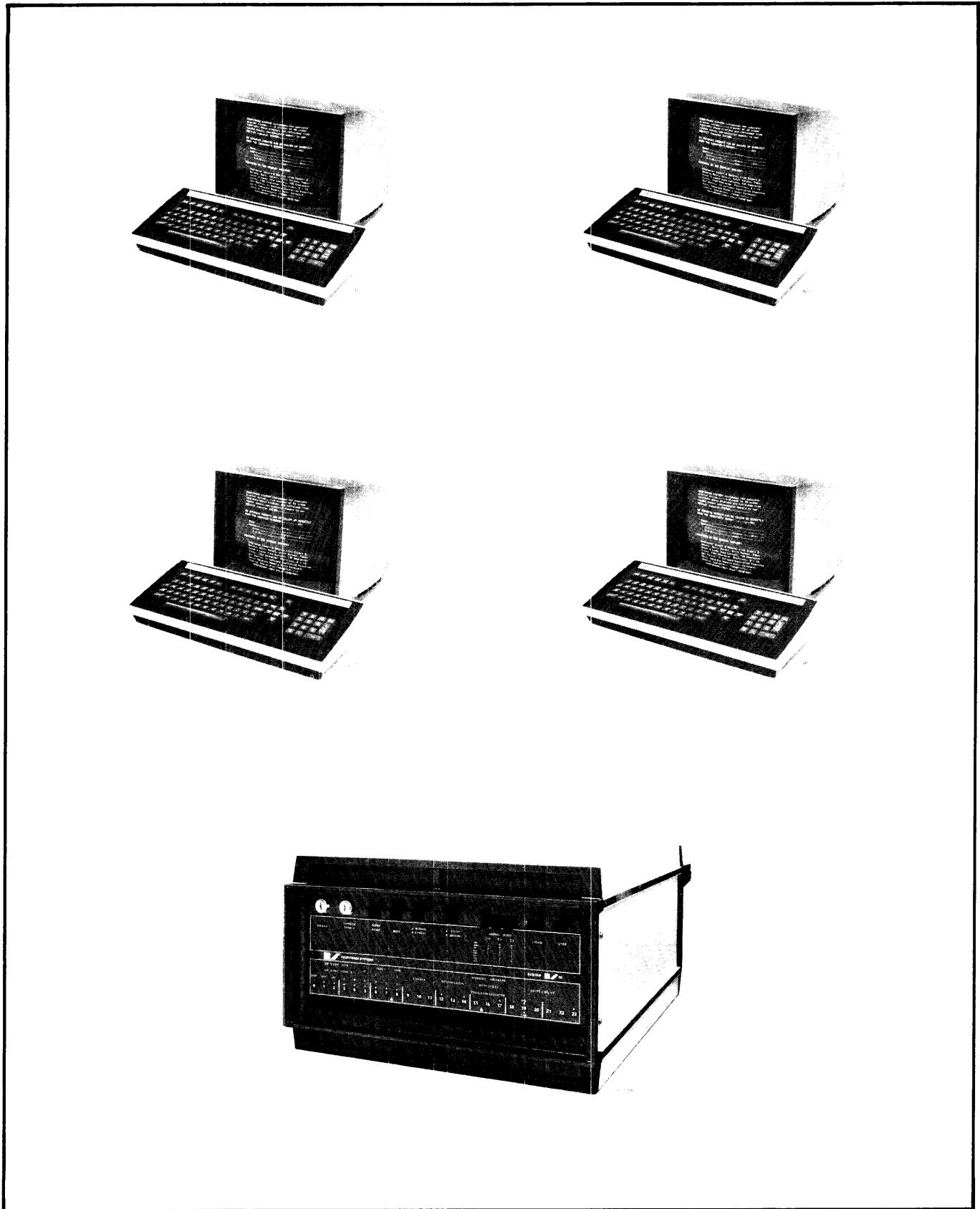
D Powers of 2 and 8

Illustrations

System IV/70 Console and Video Display Terminals	iv	6-1 Peripheral Unit I/O Structure	6-1
2-1 Simplified Block Diagram of System Architecture	2-1	6-2 Select Word and Buffer Address Word Formats	6-3
4-1 Logical and Arithmetic Compare	4-8	8-1 System IV/70 Assembler Language Coding Form	8-3
4-2 Branching to Subroutines	4-12	8-2 Assembler Output Listing	8-5
5-1 Character Byte Control, Shift Count, and Linkage Tables	5-4	9-1 Console Controls and Indicators	9-2
5-2 Character Manipulation Instruction Operands	5-5	9-2 Magnetic Tape Drive Controls and Indicators	9-7
5-3 Character List Processing Instructions	5-12	A-1 System IV/70 Display Characters	A-4

Tables

3-1 Notation and Special Usages	3-1	4-1 Condition Codes for Logical and Arithmetic Comparisons	4-9
3-2 Significance of Modification Field	3-3	5-1 Numeric Comparison Results	5-16
3-3 Example of Indexing and Indirect Addressing	3-4	7-1 Dedicated Interrupt Locations	7-1
3-4 Dedicated Memory Locations	3-6	9-1 Control Panel Controls and Indicators	9-3
3-5 Programmer-Addressable Registers	3-7		



System IV/70 Console and Video Display Terminals

Section 1

Introduction

System IV/70 is a computer-based data display and processing system designed for data entry and retrieval to and from data bases and computing systems. It interfaces with IBM 360/370 Systems locally or remotely through System IV/70 foreground communications packages without requiring the modification of IBM software. Subsets of this package completely simulate IBM 2848/2260 and 3270 data station complexes.

The System IV/70 consists of a Processing Unit, up to 32 Video Display Terminals, and peripheral devices.

The System IV/70 Processing Unit is a character-oriented medium-scale computer with a 2.0 microsecond cycle time and semiconductor main frame memory. Two units are available, the 7001 and 7002. The 7001 has a memory size expandable from 12K to 24K 8-bit bytes in 6K byte increments; the 7002 has a memory size of 48K, 72K, or 96K bytes. Both are addressable to 96K bytes. Under software control, parity is calculated on every memory write and checked on every read.

Both the 7001 and 7002 have a repertoire of 113 machine instructions including decimal arithmetic, binary fixed-point and floating-point arithmetic, translate-test, supervisor trap, register-to-register, interrupt control, list processing with push-pop stacks, and variable length character string manipulation. The Decimal Option expands the 7002 instruction set to 119 instructions. Representative operation speeds are:

- Character move 40 μ s + 2.7 μ s/byte
- Character compare 28 μ s + 4 μ s/byte
- Decimal add or subtract 36 μ s + 5.3 μ s/byte
- Binary add, subtract, or
compare 16 μ s/24 bits

The Computer Input/Output structure includes eight I/O channels, each of which may service up to 64 devices, and eight levels of nested hardware priority interrupt. All types of I/O transfer are handled with a single I/O instruction.

A maximum I/O rate of up to 375,000 bytes/second may be reached for block transfers, and up to 39,000 bytes/second for interrupt system transfers.

Each Video Display Terminal consists of a Video Display Unit (screen) and a separate Keyboard Unit. The Video Display Unit offers up to 1944 characters per screen in formats of 48 or 81 characters per line with 6, 12, or 24 lines per screen. The character set is augmented ASCII with 7 x 9 dot matrix. All video display is under computer control; refresh and display of keyboard-entered or program-generated characters are automatic. Standard features include split screen, protected display areas, and transmission of selected display data, all under software control. The Keyboard Unit produces 173 character codes including control and function codes. A standard adding machine keyboard is also included. Optional Keyboards offer special keytops for particular applications. Standard edit capability includes character insert and delete, line insert and delete, roll up/down, tab, erase, and ten cursor control functions, all under software control.

Standard peripherals include removable cartridge disc drives, asynchronous and bisynchronous communications interfaces, IBM-compatible magnetic tape drives, high-speed printers, and a card reader. Standard software packages include foreground video-display control packages including an advanced key-to-disc data entry system, a 2260/2848 simulator package, terminal communications software, disc operating system with sort and other utility programs, and a video-oriented COBOL compiler.

The 7002 Processing Unit offers all the proven features of the standard model 7001 with larger memory size and extra features. The 7002 is available with a standard 48K byte memory or an optional 72K or 96K bytes. Other optional features include 81 character by 24 line video displays (which may be intermixed with 81 by 12 displays), a lightpen for menu search on the video display, dual intensity screens with software control of intensity and blanking, and an audible keyboard alarm under software control.

Section 2 System Organization

INTRODUCTION

The System IV/70 Processing Unit is organized into Control Logic, Arithmetic Logic, Main Memory, the Input/Output system, and the Video Output Circuits. See Figure 2-1 for this configuration.

CONTROL LOGIC

The Control Logic initiates and controls all functions related to implementation of the computer program instructions. All these functions take place under control of a microprogram, which is stored in read only memory and which generates control signals for all the subsystems of the computer. These signals control inputting (both through peripherals and the control panel), processing, testing, storing, and outputting of data and instructions.

ARITHMETIC LOGIC

The Arithmetic Logic performs all data processing functions, under control of the microprogram. Processing of numeric, character, and logical data; setting of condition codes; and generation of addresses are among the functions of this subsystem. The Arithmetic Logic interacts with the Control Logic in the performance of data processing functions. The eight programmer-addressable working registers (R0, R1, RP, RA, RB, X1, X2, and X3) are contained in the Arithmetic Logic.

MAIN MEMORY

The Processing Unit's main storage consists of large-scale-integrated circuit random-access memory. The memory is all directly addressable by the Control Logic, which controls data transfer in and out of memory.

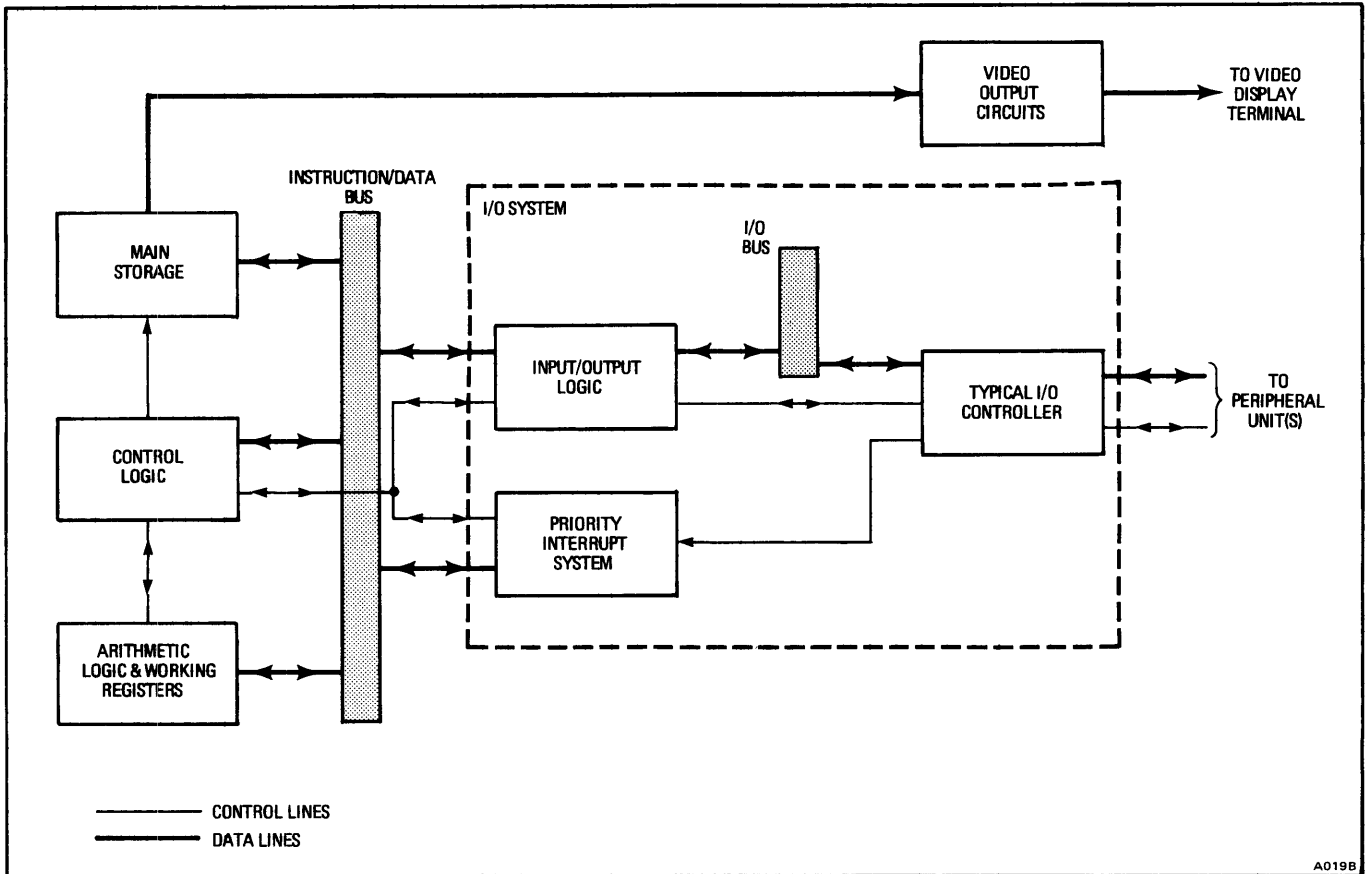


Figure 2-1. Simplified Block Diagram of System Architecture

INPUT/OUTPUT SYSTEM

The I/O system consists of the priority interrupt system, the I/O logic, the I/O controllers, and the peripheral units.

Priority Interrupt System

A nested priority system is provided with eight levels of priority and 64 chained unit addresses within each level. Each level of interrupt is connected to an I/O channel of the same number; a unique memory address is also accessible from each level to facilitate software processing of input and output over the various channels. The priority interrupt levels are truly nested, in that a level of lower priority may be interrupted during its processing by one of higher priority. These interruptions do not upset the processing of any level or of the background program. Section 5 discusses the priority interrupt structure in detail.

I/O Logic

The I/O logic responds to the I/O controllers, the Control Logic, and the interrupt system to switch I/O functions and channel and unit addresses in an orderly manner. Data to and from the I/O controllers is channeled through the I/O logic.

I/O Controllers

Each I/O device interfaces with a controller, which performs buffering, switching, and serial/parallel processing to relate the peripherals to the I/O interface. Controllers also generate interrupt signals to initiate data transfers. Each controller is designed to interface its device or devices to the computer; up to 18 controllers can be used with one System IV/70 Processing Unit.

Peripheral Units

The computer can interface (using the appropriate con-

troller) with any conventional input/output device, being fast enough to handle any widely-used data rate and flexible enough for any format. In addition, a 173-character code keyboard is used with each video display to form a highly flexible two-unit terminal. The keyboard interfaces with its own controller, which can service numerous keyboards.

VIDEO OUTPUT CIRCUITS

The Video Output Circuits receive information stored in Main Memory every other memory cycle as the result of a continual scanning process which cycles through all of memory. However, only selected areas of the memory are gated through the Video Output Circuits to be displayed at the video terminals. The addresses of these locations are listed under the table, "Dedicated Memory Locations" in Section 3.

For the 7001 Processing Unit, the first four 3072-byte blocks of memory can support up to eight 24-line, 48 character-per-line screens or eight 12-line, 81 character-per-line screens. For the 7002 Processing Unit, the first 16 3072-byte blocks of memory can support up to 32 24-line, 48 character-per-line screens or 16 24-line, 81 character-per-line screens. Using fewer lines per screen allows up to four screens to be supported by each video display area up to a maximum of 32 screens per system.

In general, the output is under computer program control: if the contents of any memory location in the dedicated area is changed, the display will be changed correspondingly. Although the software assembler and loader for System IV/70 are designed to facilitate relocatable programming, they are designed so that absolute code may be included with relocatable code. Thus, programmed transfers of data into the dedicated areas will automatically display data for the user.

Section 3

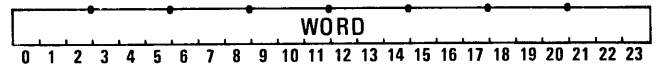
Machine Language Programming

INTRODUCTION

This section covers machine language programming in detail. Topics are Formats, Instructions, Hardware Organization, Programming for the Video/Keyboard, and Instruction Descriptions. "Formats" deals with data and instruction formats, emphasizing the formats needed by the machine language as contrasted with assembly language formats. For the assembly language see Section 8. "Instructions" describes the various fields used in instructions. "Hardware Organization" describes hardware details needed by the machine language programmer: dedicated addresses, specialized usages, constant and variable registers, and the like are emphasized. "Instruction Descriptions" explains the format used to describe the machine language instructions. The instructions themselves are covered in the following four sections. Table 3-1 defines notation used in specialized or conventional ways in this discussion.

FORMATS

Formats for data and instruction words are based on the computer's 24-bit word with the bit-positions being numbered from 0 (leftmost or most significant) through 23 (rightmost or least significant).



This information format is used to represent both data and instructions. Instructions always occupy a single 24-bit word; numeric data may occupy single, double, or triple word formats; and character data may be manipulated in blocks of up to 256 words. For readability and convenience, the bits of a word are sometimes marked off in groups of three and expressed in octal digits. The 24-bit word then becomes eight octal digits.

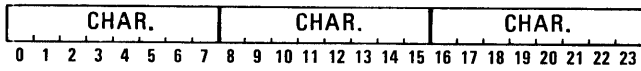
Table 3-1. Notation and Special Usages

NOTATION	MEANING
X	A symbol or name specifying a register or memory location.
[X]	The contents of the memory location or register specified by X. [X] is a number and may specify a memory location or a register. Read [RA] as "the contents of RA."
[RA] ₉₋₂₃	Bits 9 through 23 of the contents of RA.
[[X]]	The contents of the memory location specified by the contents of the memory location or register specified by X. Read [[RA]] as "the contents of the location specified by the contents of RA."
→ or ←	Replacement symbol. Read EA → [RA] or [RA] ← EA as "the effective address replaces the contents of RA" (or "the effective address is loaded into RA").
∩	"Intersection" or restrictive operator. Also called logical AND; equivalent to • in some notations.
∪	"Union" or permissive operator. Also called logical OR; equivalent to + in some notations.
⊕	XOR or differential operator. Also called exclusive OR; equivalent to ⊕ in some notations.
\overline{X}	The inverse of X; read "not X" or "X-bar."

Data Formats

CHARACTER DATA

Characters, including decimal numbers, are represented by 8-bit bytes. Three characters can be packed to a single computer word.



The internal character code used is ASCII and is shown in Appendix A.

LOGICAL DATA

Logical data is represented by full 24-bit words. The logical operations treat all the bits of the word in the same manner, as contrasted with arithmetic operations which treat the most significant bit (bit 0) as a sign bit.



ARITHMETIC DATA

All arithmetic data is represented in 2's complement form with bit position 0 as the sign bit; 0 means positive and 1 negative. Therefore, in positive numbers ones are significant bits and in negative numbers zeros are significant bits. Arithmetic shifting operations do not affect the sign bit: these operations shift around the 0 bit. Arithmetic data is of two kinds: (1) fixed point or integer, and (2) floating point or fraction-and-exponent.

Fixed Point Data

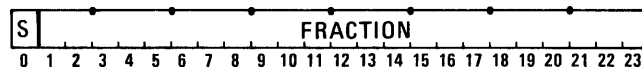
Fixed-point numbers are stored as 23-bit integers with the binary point assumed to be to the right of the least significant bit. The computer operates on these numbers arithmetically in a two's complement number system. Each 23-bit number has the equivalent precision of just under seven decimal digits; i.e., from -2^{23} ($= -8,388,608$) to $+2^{23} - 1$ ($= 8,388,607$).

Floating Point Data

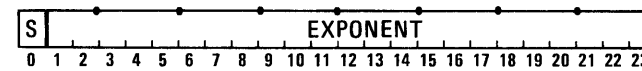
The computer accommodates two number formats for floating point arithmetic: standard and extended. Both formats consist of a fraction (or mantissa) and an exponent (power of two multiplier, or characteristic). The arithmetic registers used to store data during the execution of each floating point instruction are noted in the description of that instruction.

STANDARD FORMAT. The number is stored in consecutive memory locations with the first word (fraction) in an even location.

First word



Second word



The fractional part of a standard (single-precision), floating point number is a 24-bit proper fraction, with the leading bit being the sign and the assumed binary point just to the left of the most significant magnitude bit. The floating-point exponent (power of two) is a 24-bit integer with a leading sign bit. The standard hardware operates on both fraction and exponent in two's complement form.

Single-precision, floating-point numbers have just under seven decimal digits of precision and a decimally equivalent exponent range of $\pm 10^{2,525,223}$ ($= 10^{2^{23}} \times \log 2$).

Single-precision, floating-point numbers are normally generated from the corresponding decimal numbers using the DCS assembler directive.

EXTENDED FORMAT. An additional 23 binary bits of fraction are added to the representation by employing a three word format which is stored in three consecutive memory locations, the first being odd.

First word (odd)



Second word (even)



Third word (odd)



Extended-precision, floating-point numbers have just under fourteen decimal digits of precision and a decimally equivalent exponent range of $\pm 10^{2,525,223}$.

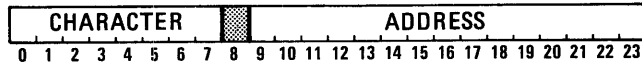
Extended-precision, floating-point numbers are normally generated from the corresponding decimal numbers using the DCD assembler directive.

SPECIAL DATA

Certain instructions use data expressed in other specialized ways. These special formats are described under the instructions that use them, and summarized below:

List Processing Instructions

These instructions use a character and an address:



Character Manipulation and Input Pack Instructions

These instructions use a format with sign bit, byte control, and count:



Instruction Format

For purposes of defining instruction formats, the word is divided into 8 octal digits (3 bits each). It is organized as follows: op code, 2 digits; modification field, 1 digit; and operand field, 5 digits.



OP CODE FIELD

This 2-digit field contains the code that designates the operation to be performed; the instruction repertoire is approximately doubled by interpreting the code differently when there is a 7₈ in the modification field.

Table 3-2. Significance of Modification Field

Contents of Mod Field	Significance
0	Not indexed and directly addressed.
1	Not indexed and indirectly addressed.
2	Indexed using X1 and directly addressed.
3	Indexed using X1 and indirectly addressed.
4	Indexed using X2 and directly addressed.
5	Indexed using X2 and indirectly addressed.
6	Indexed using X3 and directly addressed.
7	Not address modifiable.†

† Note that the hardware does not allow for indirect addressing when X3 is used for indexing.

MODIFICATION FIELD

This octal digit aids in designating the operation to be performed as described above, and designates the type of address modification (if any) as shown in Table 3-2. Address modification includes indexing and indirect addressing. Indexing can be performed using any of the three machine index registers X1, X2, or X3. See "Memory Reference Instructions" for details.

OPERAND FIELD

This field specifies the information that the programmer must supply for proper execution of the particular instruction. The five octal digits may include memory reference (reference address), shift count, register control, and byte control, depending on the particular instruction. Since an address is 15 bits in length, instructions with an address will not have any other operand information; instructions that contain an address have a type 1 format. All other instructions, except decimal option instructions, have type 2 formats. For full discussion of the operand field for any instruction, see the discussion of that instruction.

INSTRUCTIONS

There are three types of instructions: memory-reference instructions (type 1 format), non-memory-reference instructions (type 2 format), and decimal option instructions.

Memory Reference Instructions

Memory reference instructions are either address modifiable or non-address modifiable. In non-address-modifiable instructions (mod field = 0 or 7₈), the reference address in the operand field is the final or effective address (EA). The contents of this address, [EA], are fetched before the instruction is executed. Since the reference address contains 15 bits, any word in memory (up to 32,768 words) may be directly addressed without the need for indexing or indirect addressing.

In address modifiable instructions (mod field ≠ 0 or 7₈) the reference address is modified by either indexing, indirect addressing, or both. The result of these address modifications is to transform the original reference address into an effective address. The effective address is defined as the final address computed for an instruction.

If both indexed and indirect addressing are specified for an instruction, indexing will be done before indirect addressing.

INDEXING

The programmer may specify any of the three arithmetic registers X1, X2, or X3 to be the index register (see Table 3-2 for mod field options). The 15 least-significant bits of the contents of this designated index register are then treated as a 15-bit displacement value.

This displacement value is added to the reference address to obtain a new address. Only the least significant 15 bits of the sum are kept. This newly developed address is called the indexed address. Indexing is designated in CODE assembly language (see Section 8) by a tag field after the address; i.e.,
LDB VALUE, X1.

INDIRECT ADDRESSING

Indirect addressing, which is limited to one level, is specified when octal 1, 3, or 5 is found in the mod field of an instruction.

If the [mod field] = 1₈, then the contents of the reference address are fetched. The address field of this newly fetched word (bits 9-23) becomes the effective address, whose contents are fetched before execution.

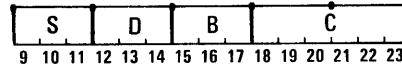
If the [mod field] = 3₈ or 5₈, the reference address is indexed to produce an indexed address; the contents of the indexed address are then fetched. The address field of this fetched word contains the effective address.

The CODE assembly program recognizes an * (asterisk) after the instruction mnemonic as the symbolic designation for indirect addressing, i.e., LDB* VALUE.

Table 3-3 illustrates the effect of indexing and indirect addressing. The operation code used is 03 for LDA.

Non-Memory Reference Instructions

The operand field may be used to specify other than address information; when no address is given, the instruction has a type 2 format. In this case the operand field is used to specify: source and destination registers, byte control, count, or nothing.† The generalized format is as follows:



where:

- S = Source register field
- D = Destination register field
- B = Byte control field
- C = Count

SOURCE AND DESTINATION FIELD

The digits in these fields specify any of the eight programmer-addressable registers located in the Arithmetic Logic. (See "Arithmetic Logic" below for a description of these registers.) They are used (with few exceptions) by instructions which operate on two registers in a source-to-destination manner; i.e., the contents of the source register replace or modify the contents of the destination register (in general, the source register will be unmodified after execution; the destination register is usually changed). Note that it is always legal to use the same register as both source and destination in an instruction. R0 and R1 are normally not specified as destination registers because they are sources of numeric constants and not storage registers. If either of these is specified as a destination, no operation will result except that appropriate condition codes will be set or reset, just as if the instruction had been executed normally. Source (bits 9-11) and destination (bits 12-14) are specified in Table 3-5.

Note that caution must be exercised in using the RP as destination: RP contains the program counter, and any change in the program counter is equivalent to a branch in the program.

† Certain instructions require no operand information. See the discussion of each instruction for this specification.

Table 3-3. Example of Indexing and Indirect Addressing

Location	Contents	Symbolic	Effect
X1	00000001		
1000	00001001		
1001	00101002		
1002	00001003		
1003	00000002		
2000	03001000	LDA 01000	[01000] = 00001001 → [RA]
2001	03201000	LDA 01000, X1	01000 + 1 = 01001; [01001] = 00101002 → [RA]
2002	03101000	LDA* 01000	[[01000]] = [01001] = 00101002 → [RA]
2003	03301000	LDA* 01000, X1	01000 + 1 = 01001; [[01001]] = [01002] = 00001003 → [RA]

BYTE CONTROL FIELD

Many instructions offer the programmer the option of specifying which bytes of the word are to be affected by the instruction. The word is broken, for this purpose, into three 8-bit bytes labeled byte 0 (most significant), byte 1 (middle), and byte 2 (least significant). Instructions that allow byte control perform the operation first, then apply byte control on the word at the time it is stored in the destination register. Thus, unselected bytes remain unchanged. Byte control is mapped into the word rather than encoded: bit 15 set means byte 0 is affected, bit 16 corresponds to byte 1, and bit 17 corresponds to byte 2.

COUNT OR SHIFT COUNT FIELD

Certain instructions employ a counter to determine the number of operations desired, e.g., shift counter or the number of significant bits in a fixed-point multiplication or division. The count field is always the least significant six bits (2-digit octal) of its instruction; thus the largest count that can be entered is $2^6 - 1 = 63_{10}$.

Note that the shift instructions are the only instructions which allow indexing or indirect addressing to be used in generating a Count Field. Thus the shift count of these instructions is modified in the same way as the memory address of other instructions, i.e., it may be direct, indirect, indexed, or both indexed and indirect, as defined by the mod field of the instruction. After modification is complete the least significant six bits of the result are used as the shift count.

Decimal Option Instructions

These instructions use special formats unique to the hardware that processes them. See the discussions of these instructions in Section 5 for details.

HARDWARE ORGANIZATION

As explained in Section 2, the computer system is logically divided into Main Storage or Memory, Control Logic, Arithmetic Logic, and the I/O System. The I/O System is covered in Section 6; this section notes salient features of the Main Storage and the Central Processing Unit, which contains Arithmetic Logic and Control Logic.

Main Storage

Main Storage on the 7001 can be 12K, 18K, or 24K bytes (4K, 6K, or 8K 24-bit words). On the 7002 it can be 48K, 72K, or 96K bytes (16K, 24K, or 32K words). The 15 bits in the address (operand) field of the instruction field allows addressing of $2^{15} = 32,768_{10}$ words of memory (ranging from 0_8 through 77777_8) without need of a base or index register or indirect addressing. These addresses are usually referred to by using a 5-digit octal number. If an attempt is

made to address non-existent memory (i.e., if too large a memory address is generated), garbage will result for the 7001. For the 7002, the last four octal digits (bits 12-23) will be used to address the first 4K words on a read; on a write the information will not be written into memory.

Odd parity is calculated on each memory write and parity is checked on each read. The parity circuits are disabled by SYSTEM RESET and controlled by the EXCT instruction (see Section 6). Control codes are 14_8 = enable parity, 15_8 = disable parity, 16_8 = select odd parity check, 17_8 = select even parity check (for diagnostic purposes only). A parity error halts the computer and activates Machine Malfunction, which is a bit displayed in position 1 of RP in MANUAL mode.

Certain memory locations are used for specified purposes and hence are considered dedicated; although the programmer may write into these areas, care must be exercised to be sure that the special functions are not disturbed. See Table 3-4.

Central Processing Unit

The CPU contains the facilities for controlling the operational sequence of instructions (the control logic function), for communicating with external devices and storage (the B3 interface function), and for performing arithmetic and logical processing of data (the arithmetic logic function).

CONTROL LOGIC

The control logic function provides the necessary means of guiding the CPU and the I/O through the operations required for execution of instructions. Implementation of system control is accomplished using random logic and a Microprogram Command Generator (MCG). The random logic performs a number of random operations and tests such as preparing the next instruction op code; handling source, destination, and byte control; and storing and testing of the status bits. The MCG controls the execution of each instruction using a stored microprogram composed of 1024 48-bit words. Each instruction is thus performed using a number of microsteps, each of which is controlled by one of the 48-bit words of the microprogram.

B3 INTERFACE

Data transfers between I/O, Main Storage, and the CPU, and within the CPU take place over B3, a bidirectional data bus that ties the various functions together. B3 also interfaces with the lights on the Control Panel: the lights display the contents of B3 at all times. When operating in the MANUAL mode, the operator views the contents of a register or a memory location via B3, for the microprogram is configured to maintain the contents of the location specified by the DISPLAY SELECT switches on B3.

Table 3-4. Dedicated Memory Locations

Octal Location	Function	Octal Location	Function
00000	Interrupt level 0	00012	Interrupt level 5
00002	Interrupt level 1	00014	Interrupt level 6
00004	Interrupt level 2	00016	Interrupt level 7
00006	Interrupt level 3	00041	Arithmetic Trap, Supervisory Trap
00010	Interrupt level 4		
7001, 48 Character/Line Video Systems†		7001, 81 Character/Line Video Systems†	
00060-00657	Video display area A	00140-00732‡	Video display area A
01060-01657	Video display area B	00740-01532‡	Video display area B
02060-02657	Video display area C	02140-02732‡	Video display area C
03060-03657	Video display area D	02740-03532‡	Video display area D
04060-04657	Video display area E	04140-04732‡	Video display area E
05060-05657	Video display area F	04740-05532‡	Video display area F
06060-06657	Video display area G	06140-06732‡	Video display area G
07060-07657	Video display area H	06740-07532‡	Video display area H
7002, 48 Character/Line Video Systems†		7002, 81 Character/Line Video Systems†	
00060-00657	Video display area 000	00140-00732‡	Video display area 00
01060-01657	Video display area 001	00740-01532‡	Video display area 01
02060-02657	Video display area 002	02140-02732‡	Video display area 02
03060-03657	Video display area 003	02740-03532‡	Video display area 03
04060-04657	Video display area 004	04140-04732‡	Video display area 04
05060-05657	Video display area 005	04740-05532‡	Video display area 05
06060-06657	Video display area 006	06140-06732‡	Video display area 06
07060-07657	Video display area 007	06740-07532‡	Video display area 07
10060-10657	Video display area 010	10140-10732‡	Video display area 010
11060-11657	Video display area 011	10740-11532‡	Video display area 011
12060-12657	Video display area 012	12140-12732‡	Video display area 012
13060-13657	Video display area 013	12740-13532‡	Video display area 013
14060-14657	Video display area 014	14140-14732‡	Video display area 014
15060-15657	Video display area 015	14740-15532‡	Video display area 015
16060-16657	Video display area 016	16140-16732‡	Video display area 016
17060-17657	Video display area 017	16740-17532‡	Video display area 017
20060-20657	Video display area 020	20140-20732‡	Video display area 020
21060-21657	Video display area 021	20740-21532‡	Video display area 021
22060-22657	Video display area 022	22140-22732‡	Video display area 022
23060-23657	Video display area 023	22740-23532‡	Video display area 023
24060-24657	Video display area 024	24140-24732‡	Video display area 024
25060-25657	Video display area 025	24740-25532‡	Video display area 025
26060-26657	Video display area 026	26140-26732‡	Video display area 026
27060-27657	Video display area 027	26740-27532‡	Video display area 027
30060-30657	Video display area 030	30140-30732‡	Video display area 030
31060-31657	Video display area 031	30740-31532‡	Video display area 031
32060-32657	Video display area 032	32140-32732‡	Video display area 032
33060-33657	Video display area 033	32740-33532‡	Video display area 033
34060-34657	Video display area 034	34140-34732‡	Video display area 034
35060-35657	Video display area 035	34740-35532‡	Video display area 035
36060-36657	Video display area 036	36140-36732‡	Video display area 036
37060-37657	Video display area 037	36740-37532‡	Video display area 037
† Video systems with 40 or 80 characters/line are achieved by programming blanks in the appropriate character positions.		‡ There are 5 unused memory locations at the end of each video line for 81 character/line systems. For example, the characters for the first line of area A occupy locations 00140-00172 while the second line of characters occupies locations 00200-00232.	

A312A

ARITHMETIC LOGIC

The arithmetic logic performs all logical, arithmetic, and shift operations on data. Of special interest to the programmer are the working registers, the condition codes, and the arithmetic trap.

Working Registers

There are eight programmer-addressable registers; six are flexible storage registers and the other two are sources of numeric constants 0_2 and 1_2 . See Table 3-5.

Condition Codes

Four condition codes (CC) are used to indicate the results of the various arithmetic logic operations. The status of the condition codes are checked using various branch and skip instructions.

The condition codes are overflow (O), zero (Z), minus (M), and carry (C). If the condition code is a 1, the condition is true; otherwise the condition code is 0 and the condition is false.

Some instructions do not alter the condition codes (see the description of each instruction for the condition codes affected). Therefore, the current value remains unchanged until altered by the appropriate instruction. An exceptional case is the overflow CC, which may be set by a number of arithmetic and shift operations, but may only be reset by the BOF instruction, which tests this CC specifically; restoration of a 1 or 0 to this CC is also possible using the BRR or BRD returns from a subroutine.† The reason for this is that an arithmetic overflow usually indicates that a data error has occurred and that all arithmetic computation from this point forward is incorrect or at least suspect. Thus, after any series of arithmetic steps that could conceivably create an overflow, a BOF should be given to allow branching to a diagnostic routine designed to clear up the problem.

Note that the MCC instruction can be used to set the condition codes on the basis of the contents of any memory location, and RCC similarly will set the condition codes on the basis of the contents of a working register. Also, the operator may change the condition codes from the console, using RP and the data keys (see Section 9 for this procedure).

OVERFLOW CC. The overflow CC permits the detection of erroneous arithmetic results that may occur during the execution of a program. The overflow CC is reset only by a BOF instruction that tests it and is set whenever the carry-outs of bit positions 0 and 1 are different from each other on a shift-left-arithmetic instruction or an addition, subtraction, or comparison. The significance of this condition for an addition or subtraction is that the number generated is too large for the register.

The significance of overflow for a shift left arithmetic is that the most significant data bit has been shifted out of the

left side of the register; significant data has been lost.

ZERO CC. The zero CC is set to 1 by certain instructions if the result of the operation was all zeros; if the result was not all zeros the CC is set to 0. All arithmetic and logical operations affect this CC.

MINUS CC. The minus CC is set to 1 by certain instructions if the result of the operation was minus; this condition is defined to be true if bit 0 of the register is a 1. If the result was not minus, the CC is set to 0.

CARRY CC. When arithmetic operations are performed, a carry can develop out of the zero position (in subtraction, a borrow is interpreted as a carry). In this event the carry CC is set to 1.

Table 3-5. Programmer-Addressable Registers

Symbol	Code	Uses
R0	0	Source of constant 0. This is a read-only register.
R1	1	Source of constant 1 (00000001 ₈). This is a read-only register.
RP	2	Program Register. Bits 9-23 contain the program counter, which holds the address of the next instruction to be executed. Bits 0-5 are used, in manual mode only, to hold and display the status bits, which include stop, machine malfunction, and the condition codes.
RA	3	Accumulator. Used for arithmetic, logic, and shift operations.
RB	4	Extended Accumulator. Used as an extension to RA in certain arithmetic operations. Available as a program scratch pad.
X1	5	Index Register 1. A hardware index register for address modification and arithmetic operations. Also available as a program scratch pad.
X2	6	Index Register 2. Same as X1 but also serves as a link register for subroutine usage by the BAL instruction.
X3	7	Index Register 3. Same as X1.

† Overflow may also be reset from the console when not under program control. See Section 9 for this procedure.

INTERPRETATION OF CONDITION CODES. In comparison instructions, an arithmetic subtraction is performed and the result is not saved; but the overflow, minus, zero, and carry condition codes are affected. See "Comparison Instructions" in Section 4 for details on the uses of the condition codes.

Arithmetic Trap

Certain arithmetic error conditions will create an arithmetic trap to Main Storage location 41₈; this location should contain an instruction which branches to a program that will test the preceding operation and discover the problem. Conditions that will create a trap include:

- An attempt to convert 4000000₈ (the largest negative number a single word can hold) to a positive number. This will occur if 4000000 is in RA at the beginning of a fixed point multiplication instruction.
- A division operation where the absolute value of the part of the numerator in RA is equal to or greater than the absolute value of the denominator [X2].
- In floating point arithmetic, any manipulation that attempts to create an exponent with absolute value greater than 2²³.
- Execution of the TRAP instruction.

PROGRAMMING FOR THE VIDEO/KEYBOARD

Video Display

The computer can accommodate up to 32 individual terminal stations, each consisting of a video screen and alphanumeric keyboard. Each display is controlled by the CPU's responses to inputs (both character codes and control codes) from the corresponding keyboard. The keyboard input character and the corresponding display outputs are shown in Appendix A. Character generation and refresh is accomplished using direct memory access output from the computer's main memory, so that memory buffers in each terminal are not required. The specific output areas that are displayed are listed in "Dedicated Memory Areas" above. The part of video display area memory that will appear on any given screen can be varied by the hardware and will be selected at the time the system is designed. The particular format selections are an integral part of the system specification and the programmer will have this information available when programming for his particular system.

Note that video-display-area memory may be used by the programmer for instructions or data, as well as for display characters. Caution should be used, however, since any instructions or data appearing in display locations will be transformed to visual information, byte by byte, causing meaningless display.

Dual intensity and hardware blanking under software control are provided as two separate options on the video display terminals for the 7002. With these options, the hardware recognizes certain patterns of bits as intensity controls and varies the brightness of the characters on the screen accordingly. With this feature, an attribute character stored in a given dedicated location affects characters that follow, with wraparound from one line to the next and from the bottom right to the top left of the screen. This feature is under control of the EXCT instruction; if EXCT with an operand of 11 (octal 13) is given or if SYSTEM RESET is pressed, no attribute characters will have any effect and the operation is identical to the 7001. The intensity will be "bright". If the DATA IV/70 system option is selected at time of manufacture and EXCT with an operand of 9 (011) is given, the 5-10-31 system goes into effect. If the 3270 system option is selected at time of manufacture and the EXCT operand is 10 (012), the 300 system goes into effect.

- *5-10-31 System.* Under this scheme, specific codes function as attribute characters. When such a character is placed on the screen, it will appear as a blank, but any data to the right will take on the video attribute specified, until another attribute character is encountered. The codes are as follows:

Code	Meaning
05 or 0205	Blank
010 or 0210	Normal
031 or 0231	Extra intensity

- *300 System.* Under this scheme, specific bit patterns are interpreted as attribute controls, with the rest of the bits in the character being subject to use by the software as the programmer requires. Thus, if any character with the specified bit pattern appears on the screen, that character will be blanked and any character to its right will take on the video attribute specified, until another character with any of the specified attribute bits is encountered. The pattern of bits is as follows:

Bit Position	0	1	2	3	4	5	6	7
Usage	1	1	x	x	y	z	x	x

A 1 indicates that the bit must be a 1 for intensity control. An x indicates don't care; the software may interpret these bits as required. The yz are the intensity control bits: 00 or 01 = normal, 10 = extra intensity, 11 = blank.

The way the cursors display is also affected by the 300 system. ASCII 032 is the usual large block cursor used with much Four-Phase supported software. ASCII 05 or 0205 are the same cursor, but function as an attribute character, blanking data to the right. ASCII 010 or 0210 are the block cursor but function as attribute characters, forcing "normal" display to

the right. Similarly, 031 and 0231 display as the block cursor and force "extra intensity" to the right. These six characters are identical in effect to other attribute characters except that they appear as block cursors instead of blanks.

The video attribute system wraps around automatically within each dedicated video area. For example, half screens or quarter screens within the same video area are all affected the same by a single attribute character occurring in any of the dedicated memory locations of that area. In general, when the variable intensity feature is used with half or quarter screens it will be necessary to control the intensities of each screen separately.

If a single attribute character is placed on the screen, its effect lingers even if it is replaced by a non-attribute character. The only restriction is that the character must remain on the screen for 1/60 of a second. If more than one attribute character is on the screen and the characters are removed, the last character scanned is the one whose effect lingers, even if it is not the last one removed. Thus, the user should wait 1/60 of a second before removing the last attribute character in a series.

Keyboard

Whenever a key on the keyboard is pressed, the keyboard controller generates an interrupt which must be serviced using the IOID instruction (see "Indirect Interrupt" in Section 7). The controller will present the device address of the keyboard terminal generating the interrupt; the program will then branch to the location specified by the IOID instruction and the device address. This location should contain an IO instruction for accepting the keyboard code from the least significant eight bits of the data bus and then storing it in a buffer area. The program must then move the character into the appropriate memory area for display (character code) or present it to a control program (control code).

INSTRUCTION DESCRIPTIONS

The following four sections describe the instructions that may be executed by the computer. Section 4 covers the conventional binary Word-Oriented Instructions. Section 5 covers the Character String Manipulation Instructions, including the Decimal Option. Section 6 discusses the Input/Output Instructions and the operation of the computer's I/O interfaces. Section 7 deals with the Priority Interrupt System Instructions. Each instruction is described separately, including all options, formats, etc. The form of each description is as follows:

Assembly Language Format

The assembly language form of each instruction comes first in the description. It contains three fields: label, mnemonic, and operand. The label field is always optional and, if used, assigns to label the Main Storage location of the

instruction. The mnemonic field contains the name of the instruction exactly as the assembly language expects to see it; if more than one mnemonic refers to the same instruction, this will be footnoted. An asterisk (*) after the mnemonic may be used to indicate indirect addressing. The operand field contains from zero to four subfields depending upon the nature of the instruction. If any operand is optional or if default options are provided, this will be noted under "Description." The contents of the operand field are numerically coded in the contents of bits 9-23 of the machine language version of the instruction (see below). If the instruction format is type 1, the operand field will contain a symbolic address reference. Indexing is indicated by adding ",Xn" where n is 1, 2, or 3 after the symbolic address reference. If the format is type 2, the operand field may contain source register, destination register, byte control, and count information.

In the operand field the following conventions apply:

Symbol	Name of Field
Expression	Symbolic address reference
S	Source register
D	Destination register
B	Byte control
C	Count

Name

Descriptive name of the instruction, in one or two lines.

Machine Language Format

Every machine language instruction consists of a 24-bit word divided into three fields: op code field, modification field, and operand field. The op code is the 6-bit code used by the computer to initiate a particular instruction; two instructions may have the same op code, but the modification field and/or the operand field will be different. The modification field will contain a one-digit octal number with the significance as previously described under "Modification Field." Thus, X in the modification field implies that address modification is allowed; 7 in the modification field means either type 2 format or no address modification, as applicable.

The Operand Field will contain up to five octal digits with significance as previously noted under "Operand Field," but expressed in a form readable by the computer. The S (Source) and D (Destination) subfields actually contain numerical register codes, addressing the working registers as follows:

Code	Register
0	R0
1	R1
2	RP
3	RA
4	RB
5	X1
6	X2
7	X3

Note that the assembly language will recognize either the name or the numerical code of the register, and that the machine language actually responds only to the binary equivalent of the octal number.

Similarly, the byte control subfield (bits 16-18) are mapped onto the three bytes of the word in a binary fashion, but written in octal:

Code	Binary Equivalent	Bytes Affected
0	000	none
1	001	2
2	010	1
3	011	1,2
4	100	0
5	101	0,2
6	110	0,1
7	111	0,1,2

The count subfield (2-digit octal) may appear in assembler language as an octal or decimal number or as an expression.

If the format of the instruction is type 1, the operand field will contain the binary address referred to by the symbolic expression in the assembly language version of the instruction. This address is usually written as a 5-digit octal integer.

Description

The instruction will be described in detail from an operational viewpoint. Sequences of operations and decision logic will be covered.

Examples

Illustrative examples, with interpretations as needed, will be given for key instructions.

Condition Codes

Condition codes are overflow, zero, minus, and carry. Instructions that alter any of these will be noted. Where a condition code is not altered, it is shown shaded in the box.

Execution Equations

Logical execution equations are provided where appropriate. \rightarrow means "replaces," [X] means "the contents of X." Thus, $EA \rightarrow [RP]$ reads "the effective address replaces the contents of RP."

Flowcharts

Flowcharts are provided for complicated instructions.

Section 4

Word Oriented Instructions

This section covers the conventional binary and word oriented instructions of the computer. Covered are word and double-word Load/Store, Fixed-Point and Floating-Point Arithmetic, Comparison, Shift, Branch and Skip, Register-to-Register, Logical, and Control Instructions.

LOAD/STORE INSTRUCTIONS

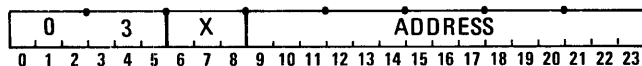
Load/store instructions move information between registers and memory. The condition codes are unaffected on all load/store instructions. All load/store instructions have type 1 formats (memory reference), and the address may be modified by indexing and/or indirect addressing.

Single Word Loads

These instructions load the contents of the effective address into a register. Note that no load RP instruction is provided; this is to prevent accidental overwriting of the program counter. The standard methods for modifying the contents of RP involve register-to-register or branching instructions.

Label LDA Expression

Load RA from EA

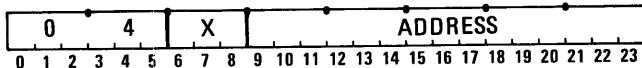


Execution Equation:

$$[EA] \rightarrow [RA]$$

Label LDB Expression

Load RB from EA

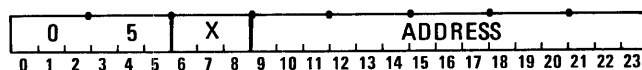


Execution Equation:

$$[EA] \rightarrow [RB]$$

Label LD1 Expression

Load X1 from EA

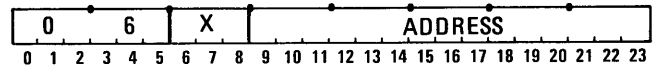


Execution Equation:

$$[EA] \rightarrow [X1]$$

Label LD2 Expression

Load X2 from EA

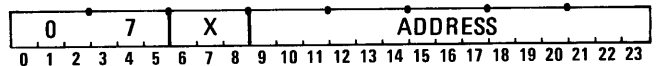


Execution Equation:

$$[EA] \rightarrow [X2]$$

Label LD3 Expression

Load X3 from EA



Execution Equation:

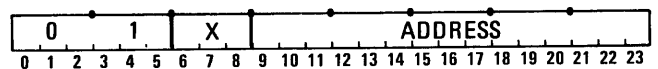
$$[EA] \rightarrow [X3]$$

Double Word Loads

These instructions load the contents of the effective address and the effective address Ored with 1 into two registers. Hence if EA is odd, the [EA] is loaded into both registers. In assembly language, the FORCE instruction is used to assign an even boundary to the starting location of such a pair of addresses. The double load instructions are commonly used to fill registers for floating point arithmetic operations, and to restore registers saved in a subroutine.

Label LDA1 Expression

Load RA from EA and X1 from EA \cup 1.

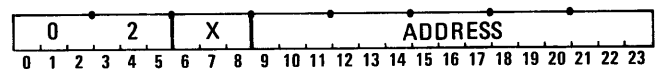


Execution Equation:

$$[EA] \rightarrow [RA]; [EA \cup 1] \rightarrow [X1]$$

Label LD23 Expression

Load X2 from EA and X3 from EA \cup 1.



Execution Equation:

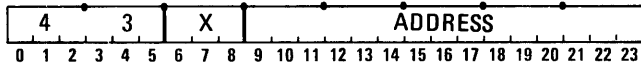
$$[EA] \rightarrow [X2]; [EA \cup 1] \rightarrow [X3]$$

Single Word Stores

Each instruction stores the contents of a working register into a memory location. The contents of the register remain unchanged. The contents of RP, RA, RB, X1, X2, or X3 can each be stored in memory using a unique instruction. Also the R0 register may be "stored," thus setting the contents of a location to zero.

Label STA Expression

Store the contents of RA in EA.

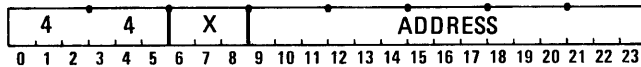


Execution Equation:

$$[RA] \rightarrow [EA]$$

Label STB Expression

Store the contents of RB in EA.

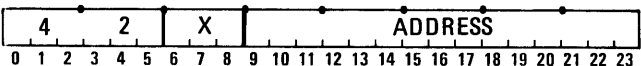


Execution Equation:

$$[RB] \rightarrow [EA]$$

Label STP Expression

Store the contents of RP in EA.

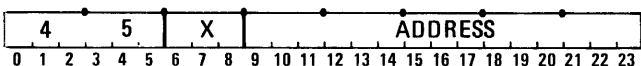


Execution Equation:

$$[RP] \rightarrow [EA]$$

Label ST1 Expression

Store the contents of X1 in EA.

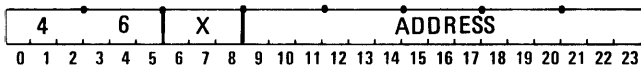


Execution Equation:

$$[X1] \rightarrow [EA]$$

Label ST2 Expression

Store the contents of X2 in EA.

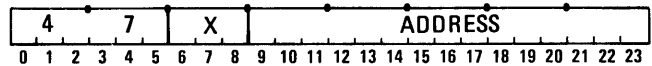


Execution Equation:

$$[X2] \rightarrow [EA]$$

Label ST3 Expression

Store the contents of X3 in EA.

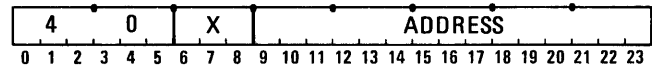


Execution Equation:

$$[X3] \rightarrow [EA]$$

Label STZ Expression

Store a word of all zeros in EA.

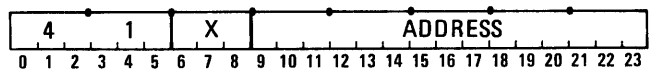


Execution Equation:

$$00000000 \rightarrow [EA]$$

Label SAM Expression

Store $[RA]_{9-23}$ in EA_{9-23} . Store $[EA]_{0-8}$ in RA_{0-8} .



Description:

This instruction may be used for modifying address locations during the execution of a program. The address part (bits 9-23) of RA is transferred into the corresponding locations in the EA, then the first nine bits (op code and modification field) of the [EA] are transferred to the corresponding position in RA.

Example:

Before Exec.	After Exec.
[RA] = 04021427	03021427
[EA] = 03000000	03021427

Execution Equation:

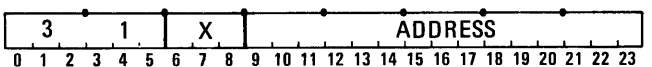
$$[RA]_{9-23} \rightarrow [EA]_{9-23}; [EA]_{0-8} \rightarrow [RA]_{0-8}$$

Double Word Stores

These instructions store two registers into memory locations EA and EA ∪ 1. If EA is odd, the second register is stored over the first.

Label STA1 Expression

Store [RA] in EA and [X1] in EA ∪ 1.

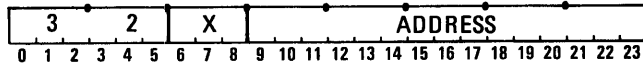


Execution Equation:

$$[RA] \rightarrow [EA]; [X1] \rightarrow [EA \cup 1]$$

Label ST23 Expression

Store [X2] in EA and [X3] in EA ∪ 1.



Execution Equation:

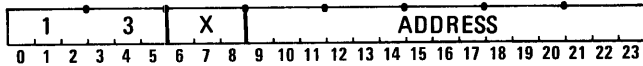
[X2] → [EA]; [X3] → [EA ∪ 1]

FIXED-POINT ARITHMETIC INSTRUCTIONS

Fixed-point arithmetic instructions perform binary addition, subtraction, multiplication, and division. The fixed-point arithmetic instructions set the condition codes depending upon the results. Depending on the instruction, the result will appear in the corresponding register or memory location. Note that the add and subtract instructions operate between memory and a register, as contrasted with the register-to-register instructions. For all addition and subtraction instructions, the format is type 1 and indexing and/or indirect addressing may be performed to generate the effective address. Multiply and Divide instructions use the type 2 format with a count field to determine the scaling† and number of significant bits in results.

Label ADA‡ Expression

Add [EA] to [RA]; result in RA.



Description:

Adds the contents of the effective address to the RA register and places the result in RA. If both numbers are of the same sign but the sign of the result is opposite, overflow has occurred and the O condition code is set. A carry can develop out of bit 0.

Example 1: Normal Add

Before Exec.	After Exec.
[RA] = 37777776	37777777
[EA] = 00000001	00000001
OZMC = 0000	0000

Example 2: Overflow

Before Exec.	After Exec.
[RA] = 37777776	40000000
[EA] = 00000002	00000002
OZMC = 0000	1010

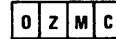
† Scaling is the process of locating the least or most significant digit in an integer. See MPY for details.

‡ The assembler will also recognize ADD as an ADA instruction.

Example 3: Carry

Before Exec.	After Exec.
[RA] = 77777776	00000000
[EA] = 00000002	00000002
OZMC = 0000	0101

Condition Code:

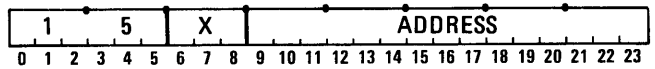


Execution Equation:

[RA] + [EA] → [RA]

Label AD1 Expression

Add [EA] to [X1]; result in X1.



Condition Code:

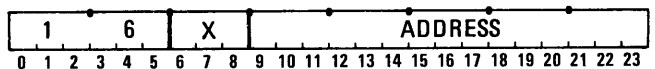


Execution Equation:

[X1] + [EA] → [X1]

Label AD2 Expression

Add [EA] to [X2]; result in X2.



Condition Code:

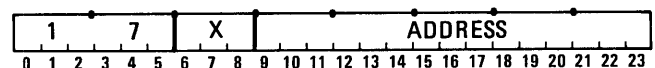


Execution Equation:

[X2] + [EA] → [X2]

Label AD3 Expression

Add [EA] to [X3]; result in X3.



Condition Code:

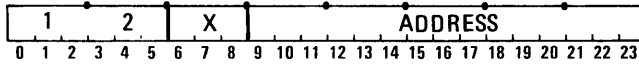


Execution Equation:

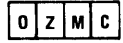
[X3] + [EA] → [X3]

Label ADM Expression

Add [RA] to [EA]; result in EA.



Condition Code:

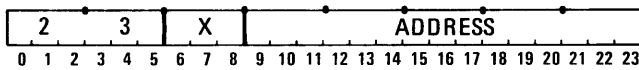


Execution Equation:

$$[RA] + [EA] \rightarrow [EA]$$

Label SBA[†] Expression

Subtract [EA] from [RA]; result in RA.



Description:

Subtracts the contents of the effective address from the contents of RA and places the results in RA. The condition codes are set in a manner appropriate for a true subtraction: if [RA] is logically less than [EA] before execution, the carry condition code will be set; otherwise it will be reset (see "Comparison Instructions"). If the two numbers are of different signs (neither zero) before execution, but the sign of the result is different from the sign of the [RA], the overflow condition code will be set.

Example 1:

Before Exec.	After Exec.
[RA] = 00000015	77777777
[EA] = 00000016	00000016
[OZMC] = 0000	0011

Example 2:

Before Exec.	After Exec.
[RA] = 37777776	40000000
[EA] = 77777776	77777776
[OZMC] = 0000	1011

Condition Code:

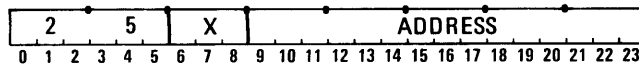


Execution Equation:

$$[RA] - [EA] \rightarrow [RA]$$

Label SB1 Expression

Subtract [EA] from [X1]; result in X1.



[†] The assembler will also recognize SUB as an SBA instruction.

Condition Code:

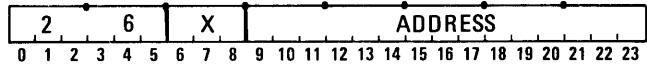


Execution Equation:

$$[X1] - [EA] \rightarrow [X1]$$

Label SB2 Expression

Subtract [EA] from [X2]; result in X2.



Condition Code:

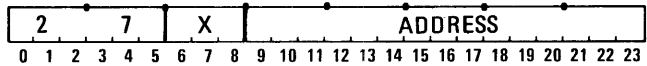


Execution Equation:

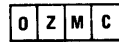
$$[X2] - [EA] \rightarrow [X2]$$

Label SB3 Expression

Subtract [EA] from [X3]; result in X3.



Condition Code:

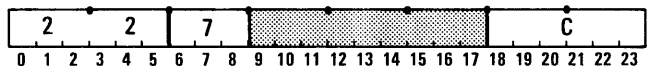


Execution Equation:

$$[X3] - [EA] \rightarrow [X3]$$

Label DIV C

Divide [RA-RB] by [X2]; quotient in RA, remainder in RB.



Description:

Performs a variable length integer division. The contents of double register RA-RB are divided by the contents of X2. The [C] least significant bits of RA, up to a maximum of 23₁₀, contain the quotient, the [C] most significant bits of the remainder appear in the [C] least significant bits of RB, and the 23-[C] least significant bits of the remainder appear in the 23-[C] most significant bits of RA. The sign bit of the quotient appears in [RA]_{23-[C]}. If no count is given, 23₁₀ will assumed.

If | [X2] | ≤ | [RA] | before execution, arithmetic trap to 41₈ in main storage will occur. The dividend can be 46₁₀ or fewer bits, scaled right, with the sign bit in RA₀ and zero in RB₂₃, which is not intended to enter into the computation. In general, the 24-[C] least significant bits of [RB] must be zero before execution or significance will be lost.

The variable length divide can be used for non-integer division, if proper alignment is kept by the program. This is shown in Examples 1 and 2. For integer division, the program can always keep correct alignment by using the method shown in Example 3: The divisor is loaded into X2 and the dividend into RA. RB is cleared, and SRAD is performed, with a count equal to the count to be used in the DIV instruction. This method will ensure that the quotient and remainder will appear, right justified, in RA and RB.

Example 1: Divide 2 in RA by 9 in X2 using DIV 027†. This is equivalent to an integer divide of 2×2^{23} by 9, or to a fixed-point divide with an assumed binary point located by the count field offset. In the latter case the binary answer is 00111000111... with the most significant bit (a zero in this case) in RA₁ and the binary point to the left of this bit.

Before Exec.	After Exec.
[RA] = 00000002	07070707
[RB] = 00000000	00000001
[X2] = 00000011	00000011

Example 2: Divide 2 in RA by 9 in X2 using DIV 020.

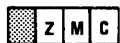
Before Exec.	After Exec.
[RA] = 00000002	00034343
[RB] = 00000000	00000005
[X2] = 00000011	00000011

Example 3: Divide 7 in RB by 3 in X2. This is a generalized method of doing integer division on quantities less than 24 bits. The quotient will appear in the least significant bits of RA and the remainder in the least significant bits of RB.

C	EQU	027	
	LDA	N7	Shift right the same number
	RCPY	R0,RB	of bits as C in DIV instruction.
	SRAD	C	
	LD2	N3	
	DIV	C	
	HLT	\$	
N7	DCN	7	
N3	DCN	3	

Before Division	After Division
[RA] = 00000000	00000002
[RB] = 00000016	00000001
[X2] = 00000003	00000003

Condition Code:

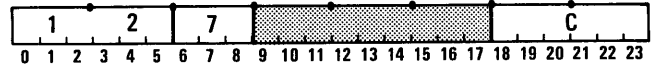


Execution Equation:

[RA-RB] ÷ [X2] → [RA], scaled right;
remainder → [RB-RA] scaled as described above.

Label MPY C

Multiply [X2] by [RA]; results in [RA-RB].



Description:

Performs a variable length integer multiplication. The contents of RA are multiplied by the contents of X2, with the results appearing in the double register RA-RB. The count field is used to position the least significant bit of the product (and/or determine the number of significant bits that will be produced). The least significant bit of the product will be found in bit position [C] - 1 of RB. Note that RB₂₃ is always zero and does not participate in the multiplication; thus the maximum useful count is 23₁₀. For a greater count, significance will be lost. If no count is given 23₁₀ will be assumed. In general, the count plus one must be equal to or greater than the number of bits in RA. RA₀ always contains a true sign bit for the product, and bits between the sign bit and the most significant bit of the product will be equal to the sign bit (i.e., not significant). Thus, the first bit in RA-RB different from the sign bit is the most significant bit of the product. A trap to 41₈ will occur if [RA] = 40000000₈ (the largest negative number) at execution.

Example 1: Multiply -10₈ by +10₈ using MPY 027†. The answer is -100₈ with the least significant bit in RB₂₂.

Before Exec.	After Exec.
[RA] = 77777770	77777777
[RB] = 00000000	77777600
[X2] = 00000010	00000010
OZMC = 0000	0010

Example 2: Multiply -10₈ by +10₈ using MPY 020. The answer is -100₈ with the least significant bit in RB₁₅.

Before Exec.	After Exec.
[RA] = 77777770	77777777
[RB] = 00000000	77740000
[X2] = 00000010	00000010

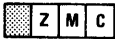
Example 3: In a normal integer multiply (C=23), the answer in RB should be adjusted using SRAD 1 after execution.

RCPY	R0,RB	Multiply 7 x 10 ₈ and scale the
LDA	O7	answer correctly.
LD2	O10	
MPY	027	
SRAD	1	
HLT	\$	
O7	DCN	7
O10	DCN	010

† A zero in front of a numerical constant indicates base 8 in assembly language.

Before MPY	After MPY	After SRAD
[RA] = 00000007	00000000	00000000
[RB] = 00000000	00000160	00000070
[X2] = 00000010	00000010	00000010

Condition Code:



Execution Equation:

$$[RA] \times [X2] \rightarrow [RA-RB]$$

FLOATING POINT INSTRUCTIONS

Floating point hardware instructions are standard. A floating point number is contained in two or three words; see "Data Formats". Standard (single precision) floating point numbers always start on an even word, extended floating point numbers on an odd word. The assembler directive FORCE lets the user insure this. The instructions perform arithmetic on standard floating point numbers. Arithmetic on extended floating point numbers is accomplished using subroutines from the Math Library.

Except for the UFA, unnormalized floating add, all floating point instructions normalize the result after execution. A normalized positive floating point number contains a 1 in bit position 1; a normalized negative number contains a zero in bit position 1. This means that the normalized number always has its most significant bit next to the sign bit.

The DCS and DCD assembler directives (see Section 8) generate constants for these computations. These instructions convert a number in modified decimal-scientific notation (absolute value of a fraction <1) to a normalized binary number to be used by the computer in the floating point instructions.

Example:

	Assembler Input	After Assembly
FL1	DCS .99999E20	00176 25536075 00177 00000103
FL2	DCS .1E+16	00200 34327724 00201 00000062
FL3	DCS -.1E1	00202 40000000 00203 00000000
FL4	DCS -.16E2	00204 40000000 00205 00000004

Register Usage

During the execution of floating point instructions, register RA and X1 form a double accumulator register consisting of:



This double register is used for both the first operand and the results. It may be loaded or stored using the LDA1 and STA1 instructions. In the case of FMP, the result also uses RB. The second operand for floating point instructions is taken from another double register consisting of X2 and X3.



This double register may be loaded or stored using the LD23 and ST23 instructions. The CDA2 instruction allows a double word copy of:

$$[RA] \rightarrow [X2]; [X1] \rightarrow [X3]$$

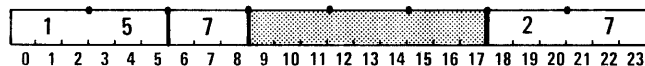
Format

The type 2 format is used for all floating point instructions. The count field must always be 23_{10} ; the assembler will furnish this count if none is given. A trap to 41_8 will occur if the exponents cannot be properly represented or the maximum negative number (40000000_8) is converted to a positive number. Note that if such a trap occurs, the registers involved (except RP) will contain intermediate results. RP will contain the address of the next instruction in sequence after the one where the trap occurred.

Instructions

Label FAD

Floating point addition.



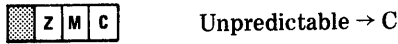
Description:

The floating point sum of the two floating point numbers in RA,X1 and X2,X3 replaces RA,X1. After execution, the sign of RA is set to the sign of the larger factor. The contents of X2 are replaced by the fraction of the factor with the larger exponent. The contents of RB are destroyed. If the fraction of the number with the larger exponent is zero, the other number will appear as the answer.

Example:

Before Exec.	After Exec.
[RA] = 25536075	25536165
[X1] = 00000103	00000103
[X2] = 34327724	25536075
[X3] = 00000062	00000062

Condition Code:

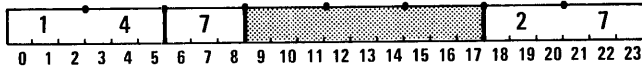


Execution Equation:

[RA,X1] + [X2,X3] → [RA,X1]
 If [X1] > [X3], then [RA] → [X2]
 Intermediate results → [RB]

Label UFA

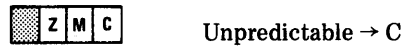
Unnormalized floating addition.



Description:

This instruction functions identically to FAD except that the result is not normalized.

Condition Code:

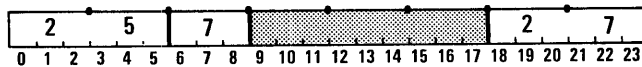


Execution Equation:

[RA,X1] + [X2,X3] → [RA,X1]
 If [X1] > [X3], then [RA] → [X2]
 Intermediate results → [RB]

Label FSB

Floating point subtraction.



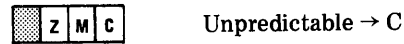
Description:

The difference of the floating point numbers in RA,X1 and X2,X3 replaces [RA,X1]. The contents of X2 are replaced by the fraction of the factor with the larger exponent. The contents of RB are destroyed. This instruction operates the same as FAD except that the subtrahend is first complemented.

Example:

Before Exec.	After Exec.
[RA] = 25536075	25536004
[X1] = 00000103	00000103
[X2] = 34327724	25536075
[X3] = 00000062	00000062

Condition Code:

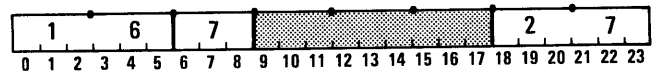


Execution Equation:

[RA,X1] - [X2,X3] → [RA,X1]
 If [X1] > [X3], then [RA] → [X2]
 Intermediate results → [RB]

Label FMP

Floating point multiplication.



Description:

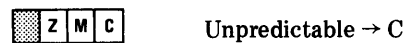
The floating point product of the two floating point numbers in RA,X1 and X2,X3 replaces [RA-RB,X1].

The fractional part of the product replaces RA and RB with the most significant part in RA. The product exponent replaces X1.

Example:

See FDV

Condition Code:



Execution Equation:

[RA,X1] × [X2,X3] → [RA-RB,X1]

Label FDV

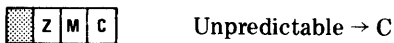
Floating point division



Description:

The quotient of the two floating point numbers, [RA,X1] divided by [X2,X3], replaces [RA,X1]. The fractional remainder replaces [RB].

Condition Code:



Example:

12_{10} is multiplied by 12_{10} to obtain 144_{10} , then divided by 12_{10} to obtain 12_{10} again.

```

LDA1   N12
LD23   N12
FMP
FDV
HLT    $
N12   DCS   .12E2
END
    
```

After Assembly

[N12] = 27777777
[N12+1] = 00000004

Before Exec.	After FMP	After FDV
[RA] = 27777777	21777776	27777777
[X1] = 00000004	00000010	00000004
[X2] = 27777777	27777777	27777777
[X3] = 00000004	00000004	00000004

Execution Equation:

$[RA, X1] \div [X2, X3] \rightarrow [RA, X1]$; Remainder $\rightarrow [RB]$

flecting the comparison without altering the contents of either the register or the memory location. The setting of the condition codes is identical to that of a subtraction. If R represents a register, a compare operation performs a subtraction to get $[R] - [EA]$ and sets the condition codes appropriately. Note that compares may set the overflow condition code; execution of a BOF instruction is required to clear this code.

The same instructions are used for both arithmetic and logical compares; the operations and results are the same, but the results are interpreted differently. The logical compare treats the most significant bit as a data bit like any other bit. The arithmetic compare treats the most significant bit as a sign bit. See Figure 4-1.

The condition code interpretations for both logical and arithmetic comparisons are listed in Table 4-1. Also listed are the conditional branch instructions that either cause a branch or do not cause a branch if the condition is true.

Example 1:

Before Exec.	After Exec.
[R] = 77777774	77777774
[EA] = 37777774	37777774
OZMC = 0000	0010

Thus, [R] is logically greater, but [EA] is arithmetically greater. The result $\overline{C} \cap \overline{Z} = 1$ implies that $[R] > [EA]$, which is true for a logical compare. The result $M = 1$ implies that $[R] < [EA]$, which is true for an arithmetic compare.

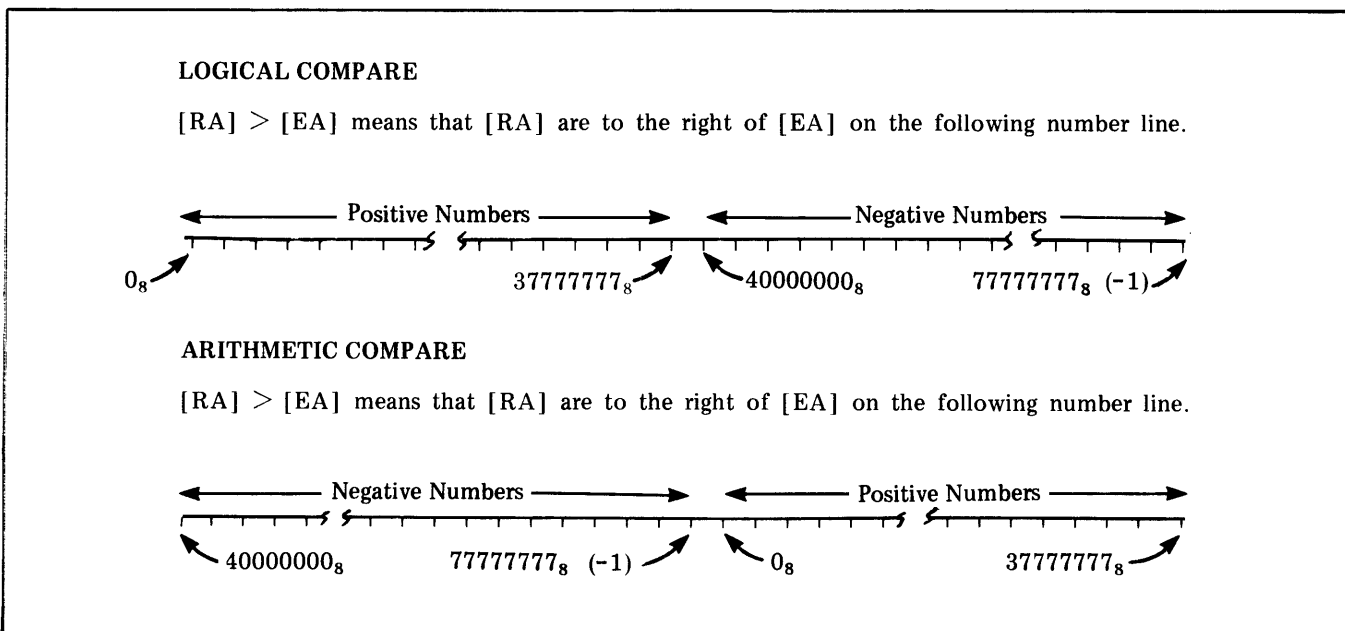


Figure 4-1. Logical and Arithmetic Compare

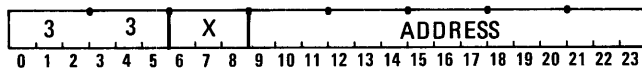
Example 2:

Before Exec.	After Exec.
[R] = 00000660	00000660
[EA] = 40000620	40000620
OZMC = 0000	1011

Thus, [R] is arithmetically greater but [EA] is logically greater. The result $C \cap \bar{Z} = 1$ implies that $[R] < [EA]$, which is true for a logical compare. The result $M \cap O = 1$ implies that $[R] > [EA]$, which is true for an arithmetic compare.

Label CPA Expression

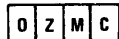
Compare [RA]:[EA] and set condition codes.



Description:

The contents of RA are compared to the contents of the EA and the condition codes set as if the operation $[RA] - [EA]$ were performed.

Condition Codes:

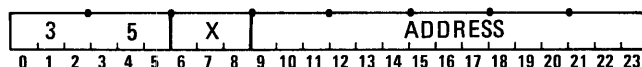


Execution Equation:

$[RA]:[EA]$

Label CP1 Expression

Compare [X1]:[EA] and set condition codes.



Condition Codes:

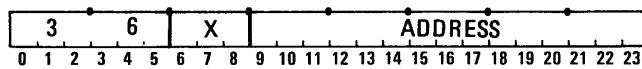


Execution Equation:

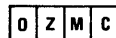
$[X1]:[EA]$

Label CP2 Expression

Compare [X2]:[EA] and set condition codes.



Condition Code:

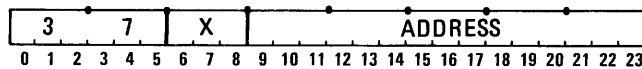


Execution Equation:

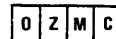
$[X2]:[EA]$

Label CP3 Expression

Compare [X3]:[EA] and set condition codes.



Condition Code:



Execution Equation:

$[X3]:[EA]$

SHIFT INSTRUCTIONS

Shift instructions operate on the RA register or on a combined RA and RB register to move data left or right

Table 4-1. Condition Codes for Logical and Arithmetic Comparisons

Condition	Logical		Arithmetic	
	Condition Code Setting	Instruction†	Condition Code Setting	Instruction†
$[R] = [EA]$	Z = 1	BZO (BNZ)	Z = 1	BZO (BNZ)
$[R] \neq [EA]$	Z = 0	BNZ (BZO)	Z = 0	BNZ (BZO)
$[R] > [EA]$	C = 0, Z = 0	BGT	M = 0, Z = 0 (and O = 0)	----
$[R] < [EA]$	C = 1, Z = 0	----	M = 1	BMI (BPL)
$[R] \geq [EA]$	C = 0	(BCR)	M = 0 (& O = 0)	BPL (BMI)
			M = 1, O = 1	----
$[R] \leq [EA]$	C = 1	BCR	----	----

† The instructions give a branch on a true condition. Instructions in parentheses allow the next instruction to be executed (do not give a branch) on a true condition.

in a variety of ways. Instructions for arithmetic, logical, and rotate (circular) shifts are provided. Note that the RRC, RLC, RCL, and RCR instructions are also provided for general inter-register shifting. See "Register-to-Register Instructions" for details.

Shift Count

Shift instructions employ the count field of the type 2 format to control the number of bits in the shift. Unlike other instructions of type 2 format, however, the count may be derived in the manner used for address modifiable instructions. In this case, the shift count is calculated exactly as is the address for an address modifiable instruction of type 1 format: first the index register is added in (if any), then the indirect address is fetched (if any) and becomes the shift count. If indirect addressing without indexing is specified, the contents of the address specified in the operand field are fetched, and become the shift count.

The contents of the mod field determine whether the contents of the operand field are an address or a count. In the discussions below, the mod field is indicated by Y or Z (Y,Z < 7): Y is even and implies count; Z is odd and implies address. Only the six least significant bits so generated are used; i.e., the count is treated modulo 64: an operand field of 74₁₀ will produce a shift count of 10₁₀.

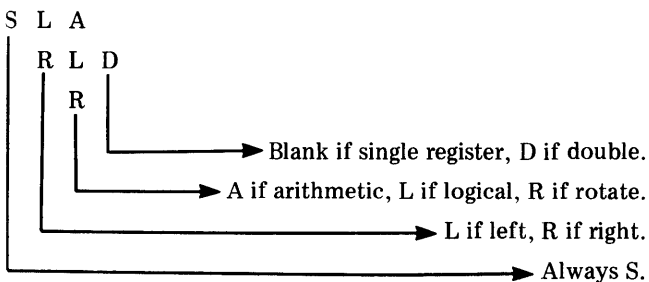
Example:

Given [X1] = 00000006

SLA 1,X1

Results in a left shift of 7.

The mnemonic name of the shift instruction indicates the direction, type, and number of registers.



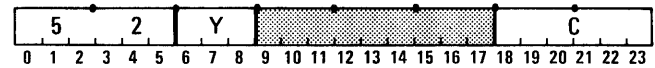
Example:

SLAD = Shift Left Arithmetic Double.

Instructions

Label SLA Expression

Shift Left Arithmetic [RA]



or



Description:

The contents of RA are shifted left [C] places. The sign position of RA does not participate in the shift. Zeros fill the vacated bit positions on the right end of register. When a bit different in value from the sign bit shifts out of RA, the overflow condition code is set.

Example:

The instruction is SLA 7

Before Exec.	After Exec.
[RA] = 76214350	43072000
O = 0	O = 1

Condition Code:

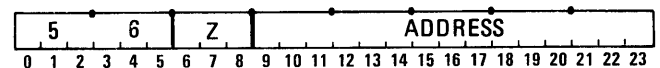


Label SRA Expression

Shift Right Arithmetic [RA].



or



Description:

The contents of RA are shifted right [C] places. The bit in the sign position of RA does not shift, but its value copies into the vacated bit positions on the left. Bits shifting past RA₂₃ are lost.

Condition Code:

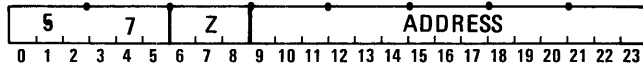
None.

Label SRAD Expression

Shift Right Arithmetic [RA-RB].



or



Description:

The contents of RA-RB are shifted right [C] places. The bit in the sign position of RA does not shift, but its value copies into the vacated bit positions on the left. Bits shifting out of RA₂₃ shift into RB₀. Bits shifted past RB₂₃ are lost.

Example:

The instruction is SRAD 12

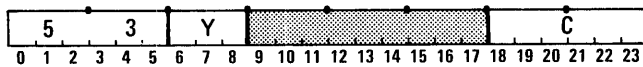
Before Exec.	After Exec.
[RA] = 63417043	77776341
[RB] = 62304110	70436230

Condition Code:

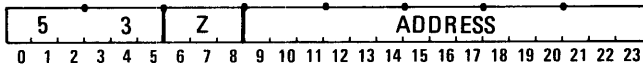
None.

Label SLAD Expression

Shift Left Arithmetic [RA-RB].



or



Description:

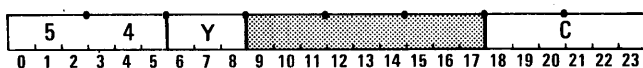
The contents of RA-RB are shifted left [C] places. The sign position of RA does not participate in the shift. Zeros fill the vacated bit positions on the right end of RB. Bits shifted out of RB₀ go into RA₂₃. When a bit different in value from the sign bit shifts out of RA, the overflow condition code is set.

Condition Code:

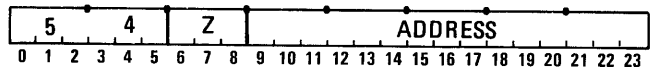


Label SRL Expression

Shift Right Logical [RA].



or



Description:

The contents of RA are shifted right [C] places. Zeros fill the vacated bit positions on the left end. Bits shifted past RA₂₃ are lost.

Example:

The instruction is SRL 17

Before Exec.	After Exec.
[RA] = 41523671	00000103

Condition Code:

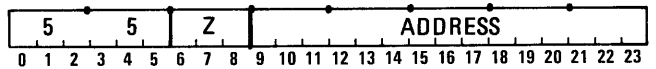
None.

Label SRLD Expression

Shift Right Logical [RA-RB].



or



Description:

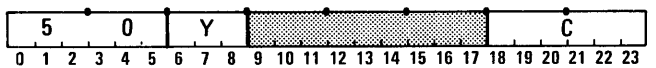
The contents of RA-RB are shifted right [C] places. The sign bit in RA and RB shifts with the rest of the number. Zeros enter RA₀ to fill vacated positions. Bits shifting out of RA₂₃ enter RB₀. Bits shifting past RB₂₃ are lost.

Condition Code:

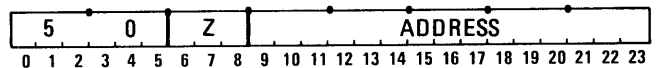
None.

Label SLR Expression

Shift Left Rotate [RA].



or



Description:

The contents of RA are rotated left [C] places. The bit in the sign position of RA shifts like any other bit. The register is treated circularly and cycles onto itself. No bits are lost. Bits shifting out of RA₀ shift into RA₂₃.

Example:

The instruction is SLR 9

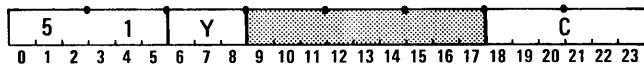
Before Exec.	After Exec.
[RA] = 20101010	01010201

Condition Code:

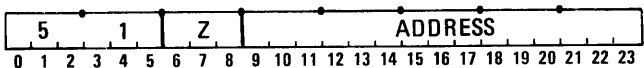
None.

Label SLRD Expression

Shift Left Rotate [RA-RB].



or



Description:

The contents of RA-RB are shifted left [C] places. The bit in the sign position of RA shifts like any other bit. The double length register is treated as if it were circular and cycles onto itself. No bits are lost. Bits shifting out of RA₀ shift into RB₂₃.

Example:

The instruction is SLRD 18

Before Exec.	After Exec.
[RA] = 26513136	36142753
[RB] = 14275363	63265131

Condition Code:

None.

BRANCH AND SKIP INSTRUCTIONS

Branch and Skip Instructions alter the normal sequence of the program by changing the program counter, which is contained in the 15 least significant bits of register RP. Branch instructions can be used to alter the program sequence, either unconditionally or conditionally. If a branch is unconditional (or conditional and the branch condition is satisfied), the instruction pointed to by the effective address of the branch instruction is the next instruction to be executed. If a branch is conditional and the condition for the branch is not satisfied, the next instruction is taken from the next location, in ascending sequence, after the branch instruction. Note that, if the branch is taken, the entire instruction is transferred into RP and the nine left-most bits of this register are therefore of little meaning to the programmer. The exceptions are the BRM instruction and in the manual mode, where the status bits are available for display and alteration in bits 0-5 of [RP].

The branch instructions used to implement the various types of subroutines are shown in Figure 4-2.

Skip instructions skip the next instruction in sequence if the condition is satisfied. Except as noted, all branch and skip instructions are type 1 format, address modifiable.

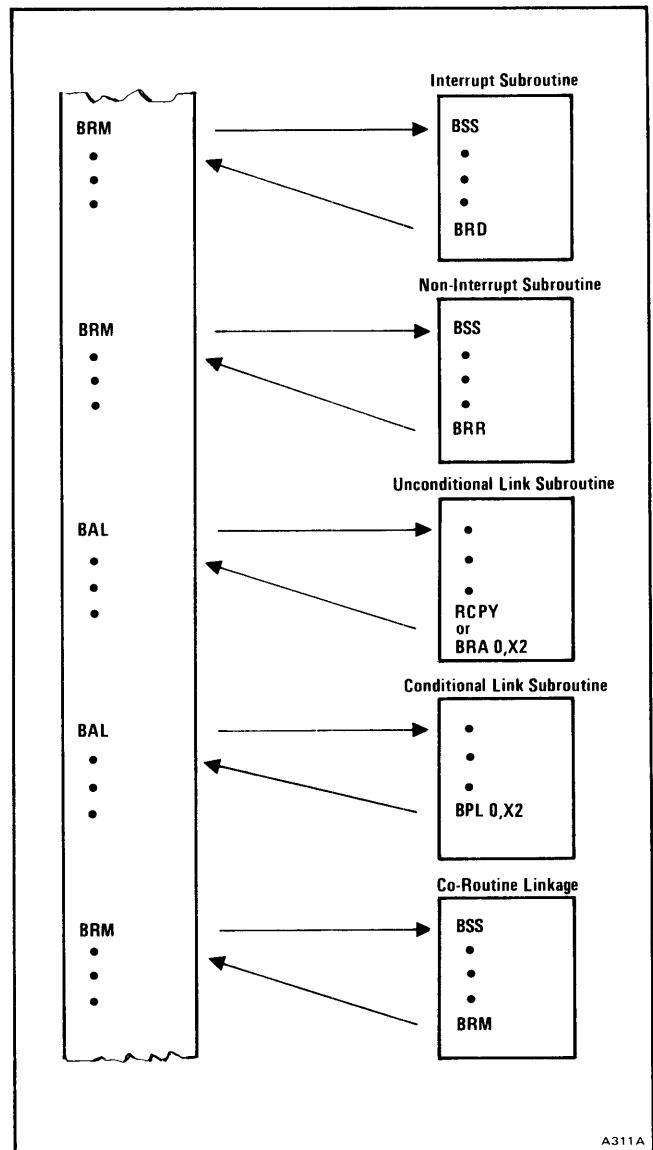
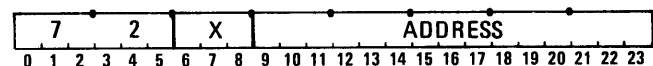


Figure 4-2. Branching to Subroutines

Unconditional Branch Instructions

Label BRA Expression

Branch to EA.



Description:

BRA causes the computer to take the next instruction from the contents of the effective address. The instruction address (after indexing and indirect address) is transferred into RP.

Condition Code:

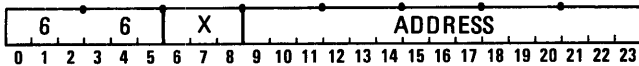
None.

Execution Equation:

EA → [RP]

Label BAL Expression

Branch to EA and link using X2.



Description:

BAL stores [RP] in X2, then replaces [RP] with the instruction address (after indexing and indirect addressing). If X2 is specified as an index, it is destroyed after the EA is calculated. This instruction is used for subroutine linkage. The value placed in X2 is the address of the instruction following the BAL, with the first nine bits indeterminate. If an interrupt signal occurs during execution of a BAL, it is not recognized until the completion of the following instruction. The next instruction can thus be used to disarm certain interrupt levels (e.g., to prevent undesirable reentrancy).

There are two kinds of return from a BAL subroutine. An unconditional return is given by RCPY X2, RP or by BRA 0, X2. A flexible return is furnished by a conditional branch (see below) to 0, X2. Note that if the [X2] are altered during performance of a BAL subroutine, the return from the subroutine will be changed.

Arguments may be passed to a BAL subroutine if data (DCN's, DCA's, instructions, etc.) are placed in the source code following the BAL instruction. This data would be fetched by performing an indexed load (e.g., LDA 1,X2 would fetch the first argument in calling sequence). The return from such a routine would come to the location following the data block; thus if six data words were transferred, the return would be via BRA 7, X2. The first location of a BAL routine must be an executable instruction, as contrasted with a BRM routine, whose first location is temporary storage.

Condition Code:

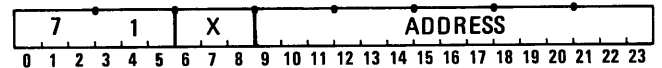
None.

Execution Equation:

[RP] → [X2]; EA → [RP]

Label BRM Expression

Branch to EA + 1 and mark place in EA.

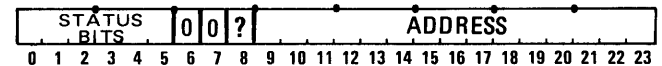


Description:

First the status bits (Stop, Malfunction, and the four condition codes) are placed in RP₀₋₅. Next [RP] → [EA] and EA + 1 → [RP]. Thus the contents of the program counter (i.e., the address of the next instruction in sequence) is kept in memory, and the next instruction is taken from the following location in memory. Return from a BRM subroutine is via a BRD (interrupt servicing) or a BRR (not interrupt), unless co-routine linkage is being implemented, in which case BRM is used on the return. If an interrupt signal occurs during execution of a BRM, it is not recognized until the completion of the following instruction. The next instruction can thus be used to disarm certain interrupt levels (e.g., to prevent undesirable reentrancy).

Arguments may be passed to a BRM subroutine if data (DCN's, DCA's, instructions, etc.) are placed in the source code following the BRM instruction. This data would be fetched by performing an indirect load (e.g., LDA* LOC would fetch the first argument in the calling sequence of a routine called using BRM LOC). If such a programming technique is used, the [EA] must be incremented by one after each argument is fetched, so that control will return to executable code, not data. Thus, the first location of a BRM routine is not normally executable (BSS 1 conventionally); this contrasts with a BAL routine, whose first location must be executable.

The exact bit format of the [EA], the word used to store the return information, is:



Where:

The address field (bits 9 through 23) are replaced by the contents of RP (the location following the BRM instruction). Note that bit 8 may be 0 or 1.

The status bit field is replaced by the setting of the Indicators as follows:

Bit Position	
0	Stop
1	Malfunction (Parity)
2	Overflow CC
3	Zero CC
4	Minus CC
5	Carry CC

Condition Code:

None.

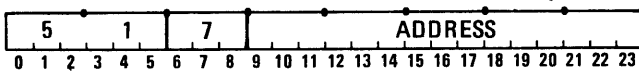
Execution Equation:

Status Bits $\rightarrow [EA]_{0-5}; [RP] \rightarrow [EA]_{9-23}$

$EA + 1 \rightarrow [RP]_{9-23}$

Label BRR Expression

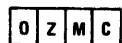
Branch Return to [EA].



Description:

BRR furnishes a return from a BRM subroutine. It replaces [RP] with [EA] and returns all four condition codes to their state before the BRM was executed. Note that no address modification may be performed.

Condition Code:

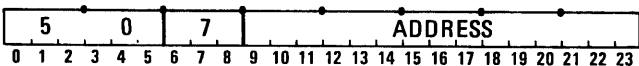


Execution Equation:

$[EA]_{2-5} \rightarrow CC; [EA] \rightarrow [RP]$

Label BRD Expression

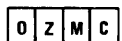
Branch Return to [EA] and Debreak.



Description:

BRD furnishes a return from a BRM subroutine that was executed as the result of an interrupt. It operates identically to a BRR except that a Debreak signal is issued to the interrupt system, thus allowing another interrupt to be serviced on the level whose servicing was just completed or any lower level. No address modification is allowed.

Condition Code:



Execution Equation:

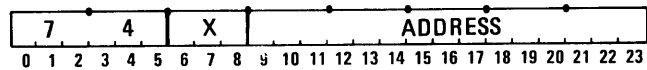
$[EA]_{2-5} \rightarrow CC; [EA] \rightarrow [RP]$

Conditional Branch Instructions

All these instructions have type 1 formats and allow address modification. They operate by testing condition codes only; not by testing the state of any register.

Label BPL Expression

Branch to EA on Plus (not minus).



Description:

If the minus condition code is reset ($M = 0$), the computer branches to the location specified by EA. Otherwise, the next sequential instruction is executed.

Condition Code:

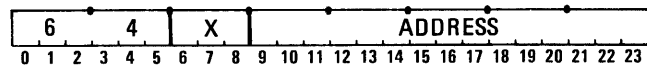
None.

Execution Equation:

If $M = 0$, $EA \rightarrow [RP]$

Label BMI Expression

Branch to EA on Minus.



Description:

If the minus condition code is set ($M = 1$), the computer branches to the location specified by EA. If not, the next sequential instruction is executed.

Condition Code:

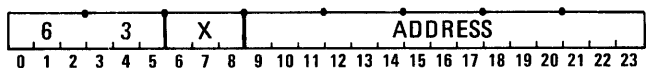
None.

Execution Equation:

If $M = 1$, $EA \rightarrow [RP]$

Label BZO Expression

Branch to EA on Zero.



Description:

If the zero condition code is set ($Z = 1$), the computer branches to the location specified by EA. If not, the next sequential instruction is executed.

Condition Code:

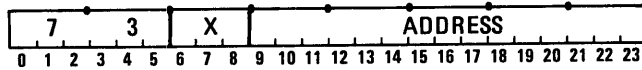
None.

Execution Equation:

If $Z = 1$, $EA \rightarrow [RP]$

Label BNZ Expression

Branch to EA on Not Zero.



Description:

If the zero condition code is reset ($Z = 0$), the computer branches to the location specified by EA. If not, the next sequential instruction is executed.

Condition Code:

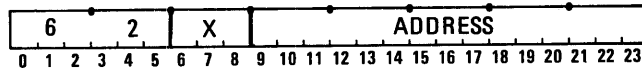
None.

Execution Equation:

If $Z = 0$, $EA \rightarrow [RP]$

Label BOF Expression

Branch to EA on overflow, reset overflow.



Description:

If the overflow condition code is set ($O = 1$) the computer branches to the location specified by EA. If not, the next sequential instruction is executed.

This instruction always resets the overflow condition code.† The procedure for resetting overflow without changing the sequence of the program is to execute $BOF \$ + 1$.

Condition Code:

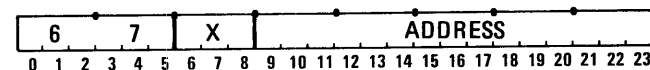


Execution Equation:

If $O = 1$, $EA \rightarrow [RP]$

Label BGT Expression

Branch to EA if logically greater than ($\bar{Z} \cap \bar{C} = 1$).



Description:

If the zero and carry condition codes are both reset ($C = 0$ and $Z = 0$), the computer branches to the location specified by EA. Otherwise, the next sequential instruction is executed.

† This is the only program instruction that will unconditionally reset the overflow CC. Zeros may also be restored to the overflow CC via the BRR or BRD instruction. The overflow CC may also be set to zero from the control panel. See Section 9.

This instruction is intended primarily for testing logical operations and index testing. The BPL and BMI instructions are used for testing general arithmetic results. The BNZ and BZO instructions are used for testing both logical and arithmetic results.

Condition Code:

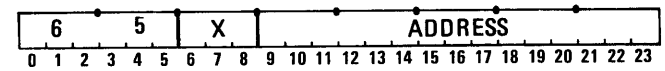
None.

Execution Equation:

If $\bar{Z} \cap \bar{C} = 1$, $EA \rightarrow [RP]$

Label BCR Expression

Branch to EA if Carry.



Description:

If the carry condition code is set ($C = 1$) the computer branches to the location specified by EA. If not, the next sequential instruction is executed. Note that this test implies "logically less than"; see "Comparison Instructions."

Condition Code:

None.

Execution Equation:

If $C = 1$, $EA \rightarrow [RP]$

Branch and Count Instructions

Branch and count instructions BC1, BC2, and BC3 are generally used in index control operations. They combine a means of incrementing an index register, testing the register for zero, and branching if the result is non-zero. If the index register is loaded with a negative count, a loop control is effected. The format is type 1 with address modification permitted.

Example:

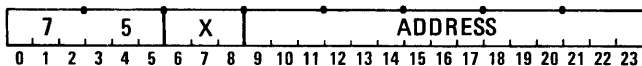
Zero out 100 locations.

LD2	M100	Get minus 100
STZ	T + 100, X2	Store Zero
BC2	\$ - 1	Increment and test
DONE	HLT	0
M100	DCN	-100
T	BSS	100
	END	

Note that this test is made on zero, not on any positive number.

Label BC1 Expression

Branch and Count X1.



Description:

BC1 increments the contents of X1 by 1. If the result is non-zero, the branch condition is satisfied and the computer takes the next instruction from the location designated by the effective address. If the result is zero, the next sequential instruction is executed.

Condition Code:

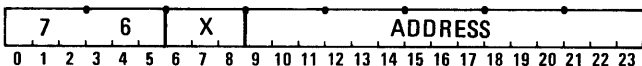
None.

Execution Equation:

$$[X1] + 1 \rightarrow [X1]. \text{ If } [X1] \neq 0, EA \rightarrow [RP]$$

Label BC2 Expression

Branch and Count X2.



Description:

Same as BC1 except that X2 is used.

Condition Code:

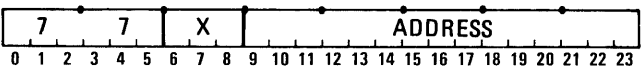
None.

Execution Equation:

$$[X2] + 1 \rightarrow [X2]. \text{ If } [X2] \neq 0, EA \rightarrow [RP]$$

Label BC3 Expression

Branch and Count X3



Description:

Same as BC1 except that X3 is used.

Condition Code:

None.

Execution Equation:

$$[X3] + 1 \rightarrow [X3]. \text{ If } [X3] \neq 0, EA \rightarrow [RP]$$

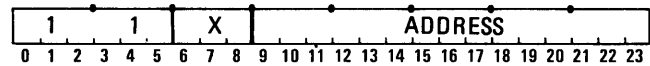
Skip Instructions

These instructions operate by adding one to the contents of the program counter if the condition specified is met. The format is type 1 with address modification.

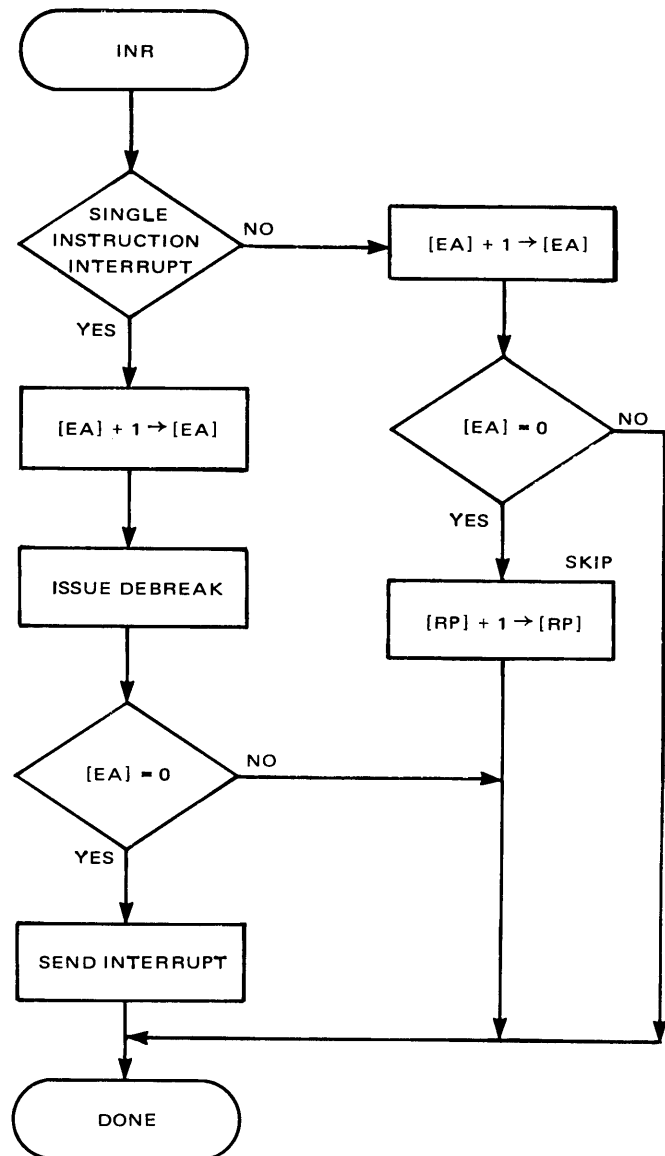
Note that these instructions alter program flow based on the contents of a memory location, without altering or referring to the condition codes.

Label INR Expression

Increment memory, skip if zero.



Flowchart:



Description:

There are two cases: single-instruction-interrupts, and others. If the INR instruction is being used to count interrupts, it

will be placed in the interrupt location for the level being serviced; otherwise it may be used to implement a counter in a memory location. Refer to the flowchart above. First a test is performed to determine whether this is a single-instruction-interrupt. If so, 1 is added to the [EA] and a debreak signal is issued to clear the interrupt being serviced. The result is then tested for zero. If zero, a signal is generated which may be wired to another interrupt level for servicing.

If this is not a single-instruction-interrupt, 1 is added to the [EA] and the result is tested for zero. If the result is zero, the skip condition is satisfied and the next sequential instruction is skipped; otherwise the next sequential instruction is executed.

Condition Code:

None.

Execution Equation:

Not a single-instruction interrupt:

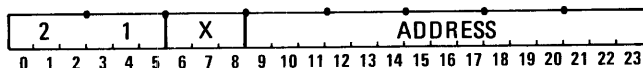
$$[EA] + 1 \rightarrow [EA]; \text{ if } [EA] = 0, [RP] + 1 \rightarrow [RP]$$

Single instruction interrupt:

$$[EA] + 1 \rightarrow [EA]; \text{ issue debreak; if } [EA] = 0, \text{ send interrupt.}$$

Label DEC Expression

Decrement memory, skip if zero.



Description:

Fetches the [EA] and subtracts one from it, then stores the result back into EA. Tests the results for zero; if zero is found, the skip condition is satisfied and the computer skips the next instruction. If not, the computer executes the next instruction in sequence.

Condition Code:

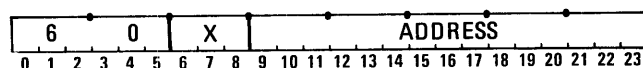
None.

Execution Equation:

$$[EA] - 1 \rightarrow [EA]. \text{ If } [EA] = 0, [RP] + 1 \rightarrow [RP]$$

Label SKZ Expression

Test memory, skip if zero.



Description:

If the contents of the effective address are zero, the computer skips the next instruction in sequence and executes the following instruction. If the contents are non-zero, the next instruction in sequence is executed.

Condition Code:

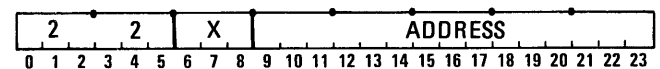
None.

Execution Equation:

$$\text{If } [EA] = 0, [RP] + 1 \rightarrow [RP]$$

Label SKN Expression

Test memory, skip if negative.



Description:

If the contents of the effective address are negative (bit 0 = 1), the computer skips the next instruction in sequence and executes the following instruction. If the contents are zero or positive, the next instruction in sequence is executed.

Condition Code:

None.

Execution Equation:

$$\text{If } [EA] < 0, [RP] + 1 \rightarrow [RP]$$

REGISTER-TO-REGISTER INSTRUCTIONS

Register-to-register instructions perform operations between a source register (S) and a destination register (D). The range of sources and destinations is the eight working registers, R0, R1, RP, RA, RB, X1, X2, and X3. Caution must be exercised in using R0 and R1 as destinations because a no-op will result except that the condition codes will be set or reset as if the instruction had been executed normally. Caution must also be exercised in using RP as a destination register, because it contains the program counter, and any change in the program counter changes the next program location. Note, however, that changing the program counter is a legitimate programming technique.

Complete facilities are provided in the register-to-register instructions for copying, rotating, clearing, adding, subtracting, complementing, and for logical operations and byte control. All instructions have the type 2 format. Except for the CDA2 and RCM2 instructions, all register-to-register instructions can use the byte store control for character selection. Byte control is applied only during data storage into the selected destination register. Therefore, if

reference is made to "before store" in the instruction description, it means after the operation has been performed but before byte control has been imposed. Byte store control is described further under "Non-Memory Reference Instructions" in Section 3.

Label CDA2

Copy double, RA-X1, X2-X3.



Description:

CDA2 copies the double accumulator RA-X1 into X2-X3. The contents of RA replace the contents of X2 and the contents of X1 replace the contents of X3. Entries in the operand field will be ignored by the assembler.

Example:

Before Exec.	After Exec.
[RA] = 23451703	23451703
[X1] = 00000004	00000004
[X2] = 00000000	23451703
[X3] = 00000000	00000004

Condition Code:

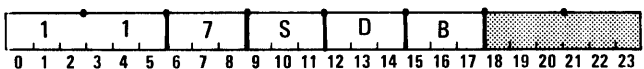
None.

Execution Equation:

[RA] → [X2]; [X1] → [X3]

Label RADD S,D,B

Register Addition, source to destination.



Description:

RADD adds the contents of the source register to the contents of the destination register. The condition codes are set according to this result. Byte store control is then effected and the result replaces the contents of the destination register. If R0 or R1 are specified as destination, no registers will be affected, but the condition codes will be set. If no byte control is specified, the assembler will furnish 7.

Example:

The instruction is RADD R1, X2, 4

Before Exec.

[R1] = 00000001

[X2] = 77777777

OZMC = 0000

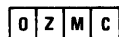
After Exec.

00000001

00177777

0101

Condition Code:

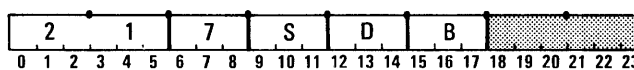


Execution Equation:

[D] + [S] → [D], selected bytes.

Label RSUB

Register Subtraction, source from destination.



Description:

RSUB subtracts the contents of the source register from the contents of the destination register. The condition codes are set according to this result. Byte store control is then effected and the result replaces the contents of the destination register. If no byte control is specified, the assembler will furnish 7. Note that RSUB R1,RP creates a closed program loop that can be cleared by moving the AUTO/MANUAL switch to MANUAL, DISPLAY SELECT to MEM, activating STEP, DISPLAY SELECT back to TIR, then AUTO/MANUAL back to AUTO.

Example:

The instruction is RSUB RA, X3, 7

Before Exec.

[RA] = 00000005

[X3] = 00000003

OZMC = 0000

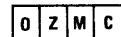
After Exec.

00000005

77777776

0011

Condition Code:

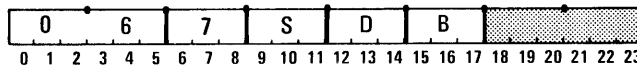


Execution Equation:

[D] - [S] → [D], selected bytes.

Label RCPY S,D,B

Copy source to destination.



Description:

RCPY copies characters from the contents of the source register and places them in the contents of the destination

register. The byte store control field selects the characters to be copied. A byte control of 7 is assumed if not furnished.

Example:

The instruction is RCPY R0, RA, 3

Before Exec.	After Exec.
[RA] = 77777776	77600000

The instruction RCPY R0, R0, 0 performs no operation.

Condition Code:

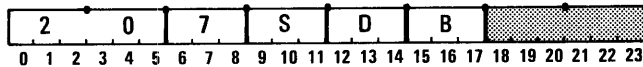
None.

Execution Equation:

[S] → [D], selected bytes.

Label RAND S,D,B

Logical AND source to destination.



Description:

RAND logically ANDs selected bytes of the source register into selected bytes of the destination register. If the corresponding bits of [S] and [D] are both 1, a 1 is placed in the corresponding bit of the destination register; otherwise a 0 is placed there. The [S] and the unselected bytes of [D] are not changed. The zero and the minus condition codes are updated based on the stored value. A byte control of 7 is assumed if not given. RAND may be used to set a bit, byte, or word to 0. If R0 or R1 is given as the destination, the condition codes are set as if the result were 0 or 1.

Condition Code:



Execution Equation:

[S] ∩ [D] → [D], selected bytes.

where:

0 ∩ 0 = 0, 0 ∩ 1 = 0, 1 ∩ 0 = 0, 1 ∩ 1 = 1

Example:

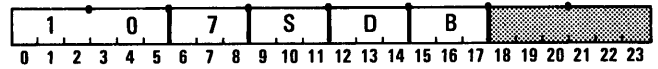
To force the [X1] even:

```
RCM2    R1, RB
RSUB    R1, RB
RAND    RB, X1
```

Before Exec.	After Exec.
[RB] = 00000000	77777776
[X1] = 22376057	22376056

Label ROR S,D,B

Logical OR source to destination.



Description:

ROR inclusively ORs the selected bytes of the source register into selected bytes of the destination register. If the corresponding bits of [S] and [D] are both 0, a 0 remains in [D]; otherwise a 1 is placed in the corresponding bit position of [D]. The [S] and the unselected bytes of [D] are not changed. The zero and minus condition codes are updated based on the stored value. A byte control of 7 is assumed if not given. ROR may be used to set a bit, byte, or word to 1. If R0 or R1 is given as the destination, the condition codes are set as if the result were 0 or 1.

Example:

The instruction is ROR RB, X2, 5

Before Exec.	After Exec.
[RB] = 25252525	25252525
[X2] = 52525252	77725377

Condition Code:



Execution Equation:

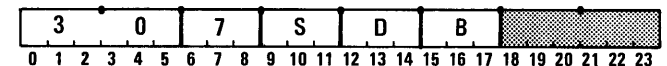
[S] ∪ [D] → [D], selected bytes.

where:

0 ∪ 0 = 0, 0 ∪ 1 = 1, 1 ∪ 0 = 1, 1 ∪ 1 = 1

Label RXOR S,D,B

Exclusive OR, source to destination.



Description:

RXOR exclusively ORs the selected bytes of the source register into selected bytes of the destination register. If corresponding bits of [S] and [D] are different, a 1 is placed in the corresponding bit position of [D]; if the contents of the corresponding bit positions are alike, a 0 is placed in the corresponding bit position of [D]. The [S] and the unselected bytes of [D] are not changed. The zero and minus condition codes are updated on the stored value. A byte control of 7 is assumed if not given. If R0 or R1 is given as the destination, the condition codes are set as if the result were 0 or 1.

Example 1:

Inverting (one's complement) two bytes. The instruction is
RXOR X2, RA, 6.

Before Exec.	After Exec.
[X2] = 77777777	77777777
[RA] = 32410616	45367216

Before Exec.	After Exec.
[RA] = 25000052	77777777
OZMC = 0000	0010

Condition Code:

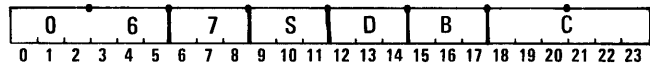


Execution Equation:

-[S] → [D]

Label RCR S,D,B,C

Copy source to destination, then rotate right.



Description:

First the contents of the source register are copied into the destination register under byte control then the contents of the destination register are rotated right by the number of positions specified in the count field. The shift is performed on all bytes of the destination. If no count is given, 0 will be assumed; if no byte control is given, 7 will be assumed. (Note that the assembler treats NOP as RCR 0,0,0,0 and RCPY as RCR S,D,B,0.)

Example 1:

The instruction is RCR R1, RA, 4, 5

Before Exec.	After Exec.
[RA] = 77777777	76003777

Example 2:

The instruction is RCR X3, RB, 5, 8

Before Exec.	After Exec.
[X3] = 05210030	05210030
[RB] = 00000000	06012400

Condition Code:

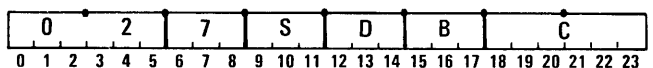
None.

Execution Equation:

[S] → [D] selected bytes; rotate [D] right [C] locations.

Label RCL S,D,B,C

Copy source to destination, then rotate left.



Example 2:

Swapping the contents of two registers without use of an intermediate location. The sequence is:

```
RXOR X1, X2, 7
RXOR X2, X1, 7
RXOR X1, X2, 7
```

Before Exec.	After Exec.
[X1] = 01234567	70707070
[X2] = 70707070	01234567

Condition Code:



Execution Equation:

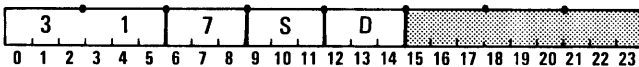
[S] ⊕ [D] → [D], selected bytes.

where:

0 ⊕ 0 = 0, 0 ⊕ 1 = 1, 1 ⊕ 0 = 1, 1 ⊕ 1 = 0

Label RCM2 S,D

Two's Complement, source to destination.



Description:

RCM2 forms the two's complement of the full contents of the source register and places the result into the destination register. The computer forms the two's complement by obtaining the one's complement and adding 00000001 to it. The results of this sum are reflected in the condition codes. Byte control is not active for this instruction.

Example 1:

The instruction is RCM2 RB, RB

Before Exec.	After Exec.
[RB] = 12134025	65643753
OZMC = 0000	0010

Example 2:

The instruction is RCM2 R1, RA

Description:

This instruction operates the same as RCR except that the direction of rotation is left.

Condition Code:

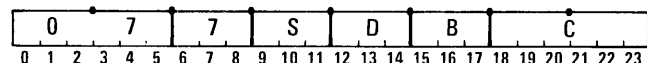
None.

Execution Equation:

[S] → [D] selected bytes; rotate [D] left [C] locations.

Label RRC S,D,B,C

Rotate right then copy, source to destination.



Description:

First the contents of the source register are rotated right (the contents of the register itself are unchanged) by the number of bit positions specified in the count field, then the rotated quantity is stored in the destination register. This instruction is particularly useful in assembling characters into words. If no shift count is given, 0 will be assumed; if no byte control is given, 7 will be assumed. Compare with RCR.

Example 1:

The instruction is RRC R1, RA, 4, 5

Before Exec.	After Exec.
[RA] = 77777777	02177777

Example 2:

The instruction is RRC X3, RB, 2, 8

Before Exec.	After Exec.
[X3] = 05200030	05200030
[RB] = 00000000	00012400

Condition Code:

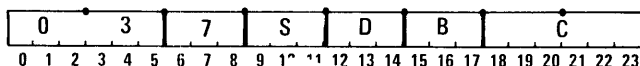
None.

Execution Equation:

Rotate [S] right [C] locations; rotated quantity → [D], selected bytes; original [S] unchanged.

Label RLC S,D,B,C

Rotate left then copy, source to destination.



Description:

This instruction operates the same as RRC except that the direction of rotation is left.

Condition Code:

None.

Execution Equation:

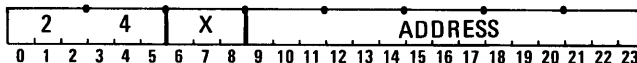
Rotate [S] left [C] locations; rotated quantity → [D], selected bytes; original [S] unchanged.

LOGICAL INSTRUCTIONS

Logical instructions operate over the entire 24-bits of two operands ([EA] and [RA]), on a bit by corresponding bit basis without regard to sign interpretation. Indexing and/or indirect addressing may be used in producing the EA. Only the zero and minus condition codes are set or reset by logical operations. In addition to the register-to-memory and memory-to-register logical operations covered here, register-to-register logical operations are also available and described under "Register-to-Register Instructions."

Label ANA† Expression

Logical AND [EA] into [RA]; results in RA.



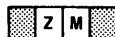
Description:

Logically ANDs the [EA] into RA. If corresponding bits of [RA] and [EA] are both 1, a 1 remains in RA; otherwise, a 0 is placed in the corresponding bit position of RA. The [EA] are unaffected.

Example:

Before Exec.	After Exec.
[RA] = 23476175	03070104
[EA] = 07070706	07070706

Condition Code:



Execution Equation:

[RA] ∩ [EA] → [RA]

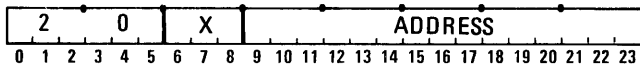
where:

0 ∩ 0 = 0, 0 ∩ 1 = 0
1 ∩ 0 = 0, 1 ∩ 1 = 1

† The assembler will also recognize AND as an ANA instruction.

Label ANM Expression

Logical AND [RA] into [EA]; results in EA.



Description:

Operates the same as ANA, except that the result of the logical AND operation appears in EA, and the [RA] are unaffected.

Condition Code:

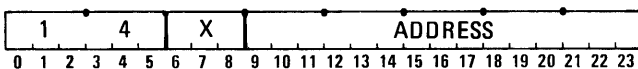


Execution Equation:

$$[RA] \cap [EA] \rightarrow [EA]$$

Label ORA† Expression

Logical inclusive OR, [EA] into [RA]; results in [RA].



Description:

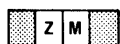
Logically ORs the [EA] into RA. If corresponding bits of [RA] and the [EA] are both 0, a 0 remains in RA; otherwise, a 1 is placed in the corresponding bit position of RA. The [EA] are unaffected.

Example:

Application of a mask for parity on three characters. [RA] before execution are ASCII "ABC"; after execution, [RA] have odd parity inserted into the first bit positions of each character.

Before Exec.	After Exec.
[RA] = 20241103	60341103
[EA] = 40100000	40100000

Condition Code:



Execution Equation.

$$[RA] \cup [EA] \rightarrow [RA]$$

where:

$$0 \cup 0 = 0, 0 \cup 1 = 1, 1 \cup 0 = 1, 1 \cup 1 = 1$$

† The assembler will recognize OR as an ORA instruction.

Label ORM Expression

Logical inclusive OR, [RA] into [EA]; results in [EA].



Description:

Operates the same as ORA, except that the result of the logical OR operation appears in EA, and the [RA] are unaffected.

Condition Code:

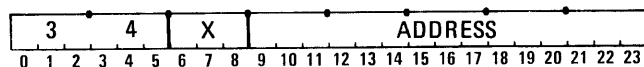


Execution Equation:

$$[RA] \cup [EA] \rightarrow [EA]$$

Label XOA† Expression

Logical exclusive OR, [EA] into [RA]; results in RA.



Description:

Logically exclusive ORs the [EA] into RA. If corresponding bits of [RA] and [EA] are different, a 1 is placed in the corresponding bit position of register RA; if the contents of the corresponding bit positions are alike, a 0 is placed in the corresponding bit position of register RA. The [EA] are unaffected.

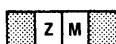
Example 1:

Before Exec.	After Exec.
[RA] = 52043716	42733712
[EA] = 10770004	10770004

Example 2: When used with an operand mask of ones, the XOA functions as a logical inversion (one's complement) of the selected bits.

Before Exec.	After Exec.
[RA] = 52043716	25734061
[EA] = 77777777	77777777

Condition Code:



† The assembler will also recognize XOR as an XOA instruction.

Execution Equation:

$$[RA] \oplus [EA] \rightarrow [RA]$$

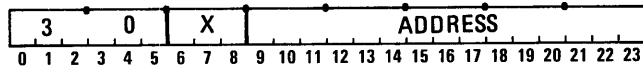
where:

$$0 \cup 0 = 0, 0 \cup 1 = 1,$$

$$1 \cup 0 = 1, 1 \cup 1 = 0$$

Label XOM Expression

Logical exclusive OR, [RA] into [EA]; results in EA.



Description:

Operates the same as XOA, except that the result of the logical exclusive OR operation appears in EA, and the [RA] are unaffected.

Condition Code:



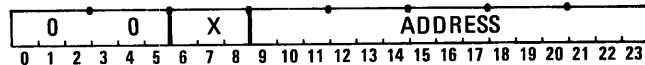
Execution Equation:

$$[RA] \oplus [EA] \rightarrow [EA]$$

CONTROL INSTRUCTIONS

Label HLT Expression

Halt operations.

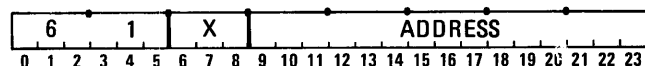


Description:

The HALT flip-flop is set. This causes the computer to enter the stop mode preventing the execution of any further instructions. To resume computation and clear the halt, the operator must either move the AUTO/MANUAL switch to MANUAL then back to AUTO, or move it to MANUAL and then activate the STEP switch. If an interrupt is received during the execution of a HLT, the interrupt will be serviced immediately after the stop mode is cleared. Note that when the halt occurs, the [RP] will be the location of HLT + 2. The [TIR] will be the next instruction after the HLT. Thus, after a HLT, moving the AUTO/MANUAL switch from AUTO to MANUAL to AUTO causes the next instruction in sequence to be executed, and execution to continue from that point.

Label MCC Expression

Test memory, set condition codes.



Description:

MCC updates the zero and minus condition codes based on the contents of the effective address. The computer accomplishes this internally by adding 00000000 to the memory location. The overflow condition code is unchanged and the carry is always set to 0.

Example:

The instruction is MCC 0100

Before Exec.	After Exec.
[0100] = 62317725	62317725
OZMC = X000	X010

Condition Code:

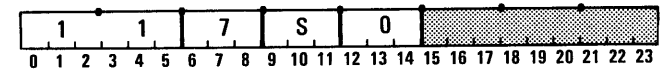


Execution Equation:

$$[R0] + [EA] \rightarrow [EA]$$

Label RCC S

Test source register, set condition codes.



Description:

RCC updates the zero and minus condition codes based on the contents of the Source Register. This is accomplished by adding R0 to the Source. Only the source must be specified. RCC uses the same op code as RADD.

Condition Code:

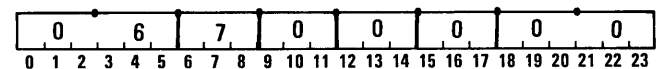


Execution Equation:

$$[R0] + [S] \rightarrow [S]$$

Label NOP

No Operation.

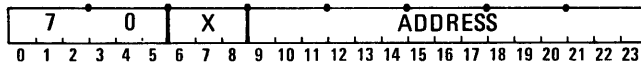


Description:

No operation takes place. The next instruction in sequence is executed. The operand field of the assembly language form should be blank.

Label XEC Expression

Execute the instruction in [EA].



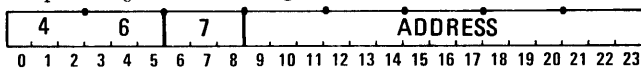
Description:

The contents of the effective address are treated like an instruction and executed in the usual manner. XEC \$ is an illegal operation which will cause the computer to hang up until SYSTEM RESET is activated.

Condition Code:
None.

Label TRAP Expression

Trap to 41_8 in main storage.



Description:

Causes a trap or supervisor call to location 41_8 , which normally contains a BRM to a routine that clears up arithmetic faults (see "Arithmetic Trap" under "Hardware Organization" in Section 3), and/or generates a supervisor call for a user defined function.

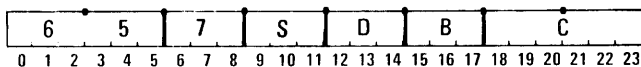
Condition Code:
None.

Execution Equation:

$$[41_8] \rightarrow [TIR]$$

Label ODD S,D,B,C

Generate odd parity on source and store in destination.



Description:

- a. The contents of the source register are read and held in the memory register.

- b. The destination register is loaded with all ones.
- c. The shift counter (C) is tested for zero; if zero the instruction ends.
- d. The counter is decremented by one and the contents of the memory register are rotated left one place.
- e. The rotated memory register is then exclusive ORed with the destination register. This result replaces all of the destination register, and the instruction loops back to step c.

The count field (C) is used to specify the number of bits over which parity is to be calculated; this is one fewer than the length of the corresponding field (e.g., for a whole word use a count of 23; for a byte use a count of 7).

Example:

Computation of parity over each of the 8-bit characters in a word and storing the result in the first bit of each character. The sequence is:

LDA	CHAR	
ANA	MSK2	Note 1
STA	WRD	Note 2
ODD	RA,RA,7,7	Note 3
ANA	MSK1	Note 4
ORM	WRD	Note 5
HLT	DONE	
MSK1	DCN	040100200
MSK2	DCN	037677577
WRD	PZE	0
CHAR	DCA	'ABC'

1. Masks out parity bit of each character.
2. Stores masked characters into WRD.
3. Generates parity bit for all positions.
4. Masks out all bits but desired parity bits.
5. Inserts parity bit into first position of each character.

Before Exec.	After Exec.
[CHAR] = 20241103	20241103
[WRD] = 00000000	60341103

Condition Code:
None.

Section 5

String Manipulation Instructions

These instructions offer various means for processing strings of 8-bit bytes. Instructions are provided for moving strings between memory blocks, for moving characters and strings between memory and RA, for translation of characters, and for processing lists of characters and words. These instructions are covered in three parts: Word and Character Manipulation Instructions, List Processing Instructions, and Decimal Option Instructions.

WORD AND CHARACTER MANIPULATION INSTRUCTIONS

These instructions furnish a means of manipulating characters (bytes) and blocks of words under programmer control. A significant application is in assembling blocks of characters for display on the video screen. For instance, up to 64 words (192 characters) of data can be assembled anywhere in memory, then transferred with one instruction into the display area for immediate display. Alternatively, a single character may be placed anywhere in a display area. Another significant application is decimal arithmetic routines that manipulate ASCII characters directly. A translate instruction is provided to implement table look-ups of character information.

No address modification is possible for these instructions.

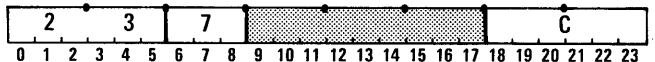
Word Move Instructions

These instructions copy a block of memory words, from one to 64 words in length, from a main storage source location into a main storage destination location. The MVE instruction moves blocks of words directly, without offset; MVL shifts a string of bytes left one byte while moving the block and MVR shifts it right one byte. Thus, MVL and MVR can be used to edit blocks into the proper locations for display or other output; for direct block transfer, the MVE instruction is much faster.

Before each of these instructions is executed, the source-block starting address minus one (plus one for MVL) must be entered into X2, and the relocation constant (i.e., the distance of the move) must be entered into X3. These instructions operate by adding the contents of X3 to the source address to obtain the destination address. No condition codes are affected. Specific byte-masking constants are required by the microprogram for the MVR and MVL instructions; they are furnished in the operands of the instructions.

Label MVE C

Move a block of words, no offset.



Description:

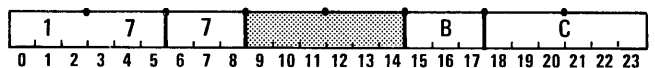
A block of words $[C] + 1$ in length is moved from a source area to a destination area. The initial source address is $[X2] + 1$ while the initial destination address is $[X3] + [X2] + 1$. The count information *must* be given. The contents of X2 will be automatically updated when execution is complete; i.e., after execution $[X2] + [C] + 1 \rightarrow [X2]$. RB is used for storage of intermediate results.

Example: A block of alphabetic characters, starting in a location identified by BLK1 is moved to BLK2.

	LD2	ADR1
	LD3	OFFSET
	MVE	5
	HLT	\$
BLK1	DCA	'ABCDEFGHIJKLMNOPQR'
BLK2	BSS	6
ADR1	DCN	BLK1-1
OFFSET	DCN	BLK2-BLK1

Label MVR B,C

Move a block of words, offset right one byte.



Description:

A block of words $[C] + 1$ in length is fetched from a source area, shifted right one byte, and stored in a destination area. The first character stored in the destination area (i.e., the last character in the $[source\ block\ starting\ address - 1]$) must be placed in the left byte of RA before execution of the MVR instruction. The last character in the source area is not stored in the destination area but appears in the left byte of RA at the end of the instruction. The initial source address is $[X2] + 1$ while the initial-destination address is $[X3] + [X2] + 1$.

Both byte and count information *must* be given. For a normal word-move, the byte control must be 3. The B register is used for temporary storage. After execution, RB contains the last word stored in the destination block, and RA contains the last word fetched from the source block,

rotated right 8 with the last character copied into the left byte. The contents of X2 will be automatically updated when execution is complete; i.e., after execution $[X2] + [C] + 1 \rightarrow [X2]$.

This instruction is convenient for insert editing of text.

Example:

```

ORG      1
LDA      SM1
RCR      RA,RA,7,8
LD2      @SM1
LD3      OFST
MVR      3,1
LDB      D+2
RCPY     RB,RA,3
STA      D+2
ORG      0140
BLK      DCA      'ZZZZZZZZZZZZZZZZ'
          DCA      'YYY'
SM1      DCA      'XXABCDEFGF'
D         EQU      BLK+1
OFST     DCN      D-SM1-1
@SM1     DCN      SM1
END      1
    
```

Result starting at 0140:

ZZZ ABC DEF GZZ ZZZ YYY XXA BCD EFG

Execution Equation:

$[RA] \rightarrow [RB]; [X2] + 1 \rightarrow [X2];$
 $[[X2]]$ rotated right 8 $\rightarrow [RA]$
 $[RA] \rightarrow [RB]$, selected bytes; $[RB] \rightarrow [[X2] + [X3]]$
 Repeat the above $[C]$ times.

Label MVL B,C

Move a block of words, offset left one byte.



Description:

A block of words $[C] + 1$ in length is fetched from a source area, shifted left one byte, and stored in a destination area. The initial source address is the highest address in the source area. The last character desired in the destination area (left byte of word in $[\text{initial source address} + 1]$) must be placed in the right byte of RA before execution of the MVL instruction. The first character in the source area is not stored in the destination area but appears in the right byte of RA at the end of the instruction. The initial source address is $[X2] - 1$ while the initial destination address is $[X3] + [X2] - 1$.

Both byte and count information *must* be given. For a normal word-move, the byte control must be 6. The B register is used for temporary storage. After execution, RB contains the last word stored in the destination block, and RA contains the last word fetched from the source block, rotated left 8 with the last character copied into the right byte. The contents of X2 will be automatically updated when execution is complete; i.e., after execution $[X2] - [C] - 1 \rightarrow [X2]$.

This instruction is convenient for delete editing of text.

Example:

```

ORG      1
LDA      SP1
RCL      RA,RA,7,8
LD2      SP1
LD3      OFST
MVL      6,1
LDB      D-2
RCPY     RB,RA,6
STA      D-2
ORG      0140
BLK      DCA      'ZZZZZZZZZZZZZZZZ'
          DCA      'YYY'
SM1      DCA      'ABCDEFGGXX'
D         EQU      BLK+2
OFST     DCN      D-SM1-1
SP1      DCN      SM1+2
END      1
    
```

Result starting at 0140:

ZZA BCD EFG ZZZ ZZZ YYY ABC DEF GXX

Execution Equation:

$[RA] \rightarrow [RB]; [X2] - 1 \rightarrow [X2];$
 $[[X2]]$ rotated left 8 $\rightarrow [RA]$
 $[RA] \rightarrow [RB]$, selected bytes; $[RB] \rightarrow [[X2] + [X3]]$
 Repeat the above $[C]$ times.

Character Manipulation Instructions

The character manipulation instructions move data in eight-bit bytes in a manner appropriate for working with character strings. The data is moved between RA and specified memory locations. There are eight character manipulation instructions; the mnemonics are constructed as follows:

- 1st letter *L* for load, *S* for store
- 2nd letter *C* for character, *P* for parallel
- 3rd letter *L* for left, *R* for right.

Load is to copy a character or string from memory into RA; *store* is to move the character or string from RA into

memory. *Character* means a single character will be loaded or stored; *load parallel* means three consecutive bytes will be moved from any arbitrary byte boundary in memory into RA (i.e., all three bytes can be in one word, or the two leftmost bytes in one word and the rightmost in the next, or the leftmost in one word and the two rightmost in the next). *Store parallel* means the word in RA will be stored into the appropriate memory location and then the pointer to that word incremented so that the next word will go into the next location. *Left* and *right* refer to the direction of the move: a left move starts with the rightmost byte or word in a string and works to the left; a right move starts with the leftmost byte or word, and works to the right. Another way of saying this is that right means working from a lower toward a higher numbered memory address; left means from higher to lower.

For these manipulations, certain constants and variables must be available to the microprogram that controls the computer. These are furnished by the programmer in the table shown in Figure 5-1, and in the other operands associated with the execution of each instruction. Since there is room in each instruction for only a single memory reference operand, the method used is to have the effective address of the instruction (i.e., the contents of bits 9-23; no address modification is allowed with these instructions) point to a word pair on an even memory boundary (FORCE 0), and have the first word of this pair point to a second word pair on another even boundary. This furnishes sufficient information for manipulating one or three byte blocks on arbitrary boundaries, using the same four operands for load and store, character and parallel operations. The following example shows the reasons for sharing the operands: it is possible to move byte strings on arbitrary boundaries, moving a character at a time until a word boundary is found in the destination area, then moving parallel words until the closing boundary is reached, at which point characters are moved again to finish the move.

The four operands are illustrated in Figure 5-2 and defined as follows: The first operand [EA] contains the pointer to the two-word table entry, explained below. The second operand [EA ∪ 1] — i.e., the contents of the next word after the pointer to the table — is a pointer (originally) to the first word in the source data block (for load) or the first word in the destination data block (for store). This pointer is updated every time a word boundary is reached (character operations) or every time the instruction is executed (parallel operations). Only the address part of the second operand is valid after execution; the left nine bits are destroyed.

The third operand [[EA]] — i.e., the word pointed at by the word stored in the effective address location — is the first word of one of three word-pairs in the Character Byte Control, Shift Count, and Linkage Tables. Thus, [[EA] ∪ 1] is the second word of the pair in the table. As shown in Figure 5-1, there are four tables of constants: one each for Load Right, Store Right, Load Left and Store Left. Each

table contains three pairs of entries, and each pair has a pointer in the first word and a byte count and shift control constant in the second word. The pointers are arranged circularly and, each time the instruction is executed, the current pointer replaces the contents of the effective address for character operations. Thus, whenever a character is taken from or put into the last byte location of a word, the data address pointer is updated (+1 for right, -1 for left) and the next operation involves the first byte (leftmost for right operations, rightmost for left operations) of the next word. The shift and byte information in the second word of each entry is arranged to enable the microprogram to perform the correct shifting and byte-masking for conventional character manipulation operations — with different table entries, different operations might be implemented. This table is standard with all Four-Phase software and is furnished in relocatable form under DOS. The labels used in these tables are constructed as follows:

Character Position in Label	Meaning
1st Character	<i>L</i> for load, <i>S</i> for store
2nd Character	<i>R</i> for right, <i>L</i> for left
3rd Character	<i>0</i> for left byte, <i>1</i> for middle byte, <i>2</i> for right byte
4th Character	Always <i>T</i> for table.

After execution of any of these eight instructions, the zero condition code is always set.

Example:

This general-purpose block move routine exercises the Character- and Parallel- Move instructions. It is the fastest method for moving arbitrary blocks of bytes without the decimal option. The calling sequence for the routine furnishes the references to the table in Figure 5-1 in the following manner: SPNT and DPNT each point to a word pair (first and second operands), the first word of which points to the appropriate table entry (third and fourth operands), and the second word to the starting address of source or destination. The table entries (third operands) are derived as follows:

Starting Byte Position	[SPNT]	[DPNT]
Leftmost	LR0T	SR0T
Middle	LR1T	SR1T
Rightmost	LR2T	SR2T

	ENTRY	MOVE	
*	BAL	MOVE	Calling sequence
**+0	DCN	LENGTH	In bytes
**+1	DCN	SPNT	Source wrd pair addr

Section 5
String Manipulation Instructions

SIV/70-11-1C

*+2	DCN	DPNT	Dest word pair addr	BZO	LP4	Skip branch if X1 < 0
*+3		RETURN	Next instruction after	BPL	LP2	
*			move done	LPR	SRCE	Move dest align part
MOVE	ST1	SAVE1	Avg cycles = 260	SPR	DEST+1	
	LDA1*	1,X2	+23*length	BRA	LP1	
	STA1	SRCE	Get table entries and	LP4	SRCE	
	LDA1*	2,X2	source and destination	SPR	DEST+1	
	STA1	DEST	addresses	BRA	RET3	
	RCPY	R0,X1		LP2	SB1	D3
	SB1	0,X2		LP3	LCR	SRCE
	BPL	RET3		SCR	DEST	Move last non-aligned
*				BC1	LP3	
LP0	SKN*	DEST	Test for word bound	RET3	LD1	SAVE1
	BRA	LP1		BRA	3,X2	Restore X1
	LCR	SRCE	Move til dest aligned	D3	DCN	3
	SCR	DEST		SAVE1	BSS	1
	BC1	LP0		SRCE	FORCE	0
	BRA	RET3		DEST	BSS	2
*				DEST	BSS	2
LP1	AD1	D3		END		

Relative Mem Address	Contents	FORCE	0	
	*		LOAD RIGHT	
00000	00000002 LR0T	DCN	LR1T	
00001	00000000	DCN	0000	
00002	00000004 LR1T	DCN	LR2T	
00003	00000410	DCN	0410	
00004	00000000 LR2T	DCN	LR0T	
00005	40000610	DCN	0610+040000000	RIGHT SHIFT
	*		STORE RIGHT	
00006	00000010 SR0T	DCN	SR1T	
00007	00000300	DCN	0300	
00010	40000012 SR1T	DCN	SR2T+040000000	NOT ALIGNED
00011	00000510	DCN	0510	
00012	40000006 SR2T	DCN	SR0T+040000000	NOT ALIGNED
00013	40000610	DCN	0610+040000000	LEFT SHIFT
	*		LOAD LEFT	
00014	00000020 LL0T	DCN	LL2T	
00015	00000300	DCN	0300	
00016	00000014 LL1T	DCN	LL0T	
00017	00000110	DCN	0110	
00020	00000016 LL2T	DCN	LL1T	
00021	40000010	DCN	0010+040000000	RIGHT SHIFT
	*		STORE LEFT	
00022	40000026 SL0T	DCN	SL2T+040000000	NOT ALIGNED
00023	00000300	DCN	0300	
00024	40000022 SL1T	DCN	SL0T+040000000	NOT ALIGNED
00025	00000510	DCN	0510	
00026	00000024 SL2T	DCN	SL1T	
00027	40000610	DCN	0610+040000000	LEFT SHIFT
00030	00000000	END		

THE COMMENTS EXPLAIN THE MEANING OF THE SIGN BIT TAGS.
Standard Four-Phase Software assumes that the tables are used exactly as shown.

Figure 5-1. Character Byte Control, Shift Count, and Linkage Tables

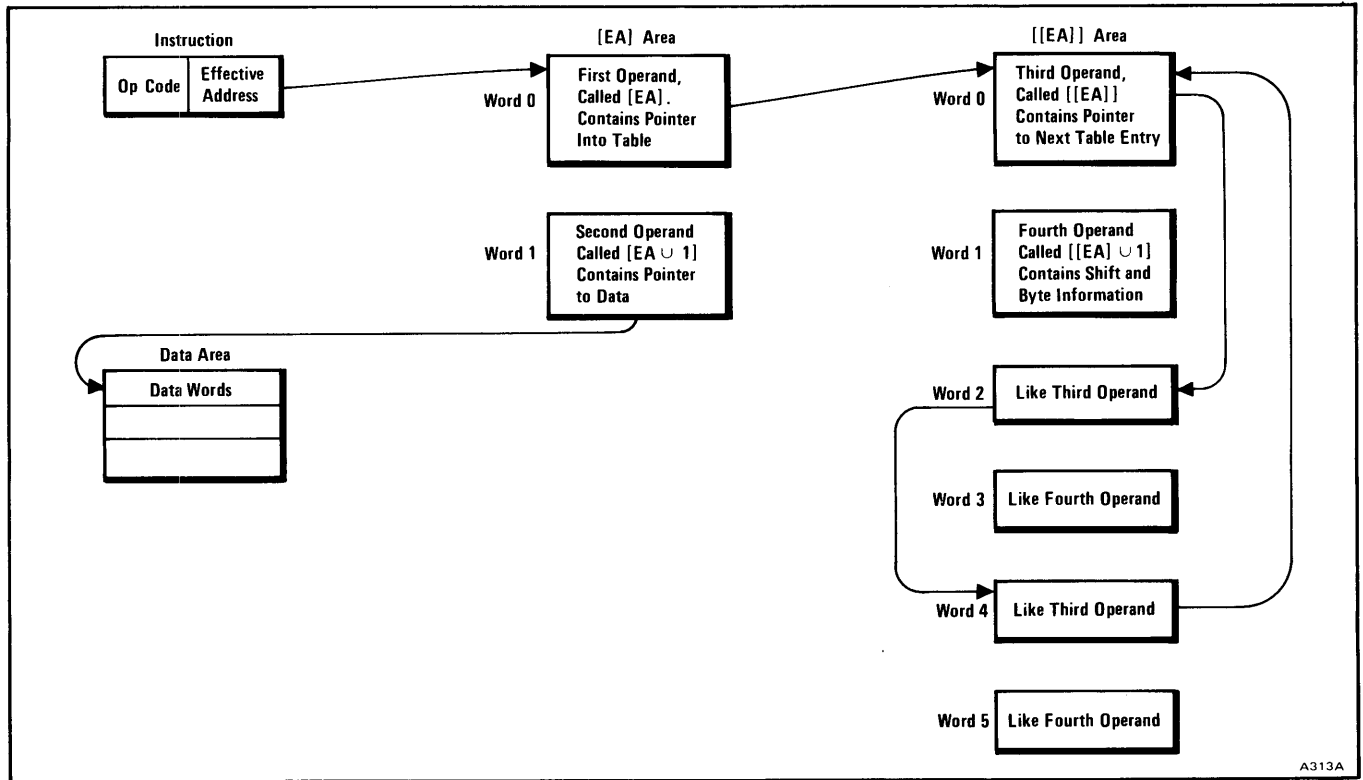
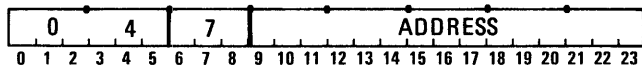


Figure 5-2. Character Manipulation Instruction Operands

Label LCR Expression

Load Character Right



Description:

LCR zeros out RA, then fetches a character out of memory from the location indicated by the second operand (see Figure 5-2). One byte of this word is copied into the leftmost byte of RA; which byte is determined by the fourth operand in the following manner:

- Bit 0 Shift direction (0 for rotate left, 1 for rotate right)
- Bits 18-23 Shift count

Thus, if the fourth operand = 00000000, the leftmost byte is taken; if 00000010, the middle byte; and if 04000010, the right byte of the word is loaded into RA. The byte controls shown in the Load Right table in Figure 5-1 are ignored. If the shift direction bit is 1 (right byte), binary 1 is added to the second operand, so that the next data will be taken from the next higher word in memory. The third operand replaces the first operand; this has the effect of moving the pointer to the next item in the circular table. RB is used for storage of intermediate results.

Example:

Storing the three bytes of a data word ([DW] = ABC) in the left byte of three consecutive memory locations ([WST] = A-, [WST + 1] = B-, [WST + 2] = C-) using the preceding character-manipulation table. Note that the [WORD ∪ 1] (i.e., DW) are incremented by one with the third execution of the LCR instruction. This allows the next three characters to be stored in the next three consecutive locations.

BEGIN	LD1	M6
LP1	LCR	WORD
	STA	WST+N,X1
	BC1	LP1
	HLT	\$
N	EQU	6
	FORCE	0
WORD	DCN	LR0T
	DCN	DW
DW	DCA	'ABCDEF'
WST	BSS	N
M6	DCN	-6
	END	BEGIN

Condition Codes:



Execution Equation:

$[[EA]] \rightarrow [EA]$

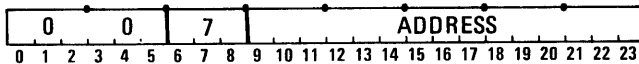
$0 \rightarrow [RA]$

$[[EA \cup 1]]$ shifted in the direction and by the number of bits designated by $[[EA]^\dagger \cup 1]$, and stored in $[RA]_{0-7}$.

If shift direction bit is 1, $[EA \cup 1] + 1 \rightarrow [EA \cup 1]$.

Label LCL Expression

Load Character Left



Description:

LCL operates the same as LCR except that one is subtracted from the second operand if the end condition is detected; this condition is shift direction bit = 0 and shift count = 0 (leftmost byte). Thus the LCL instruction is designed for moving from right to left (higher to lower memory addresses) in memory, whereas LCR is designed for going from left to right (lower to higher addresses). Note that the first operand must start with LR0T for LCR and with LL2T for LCL when starting on a word boundary.

Condition Codes:



Execution Equation:

$[[EA]] \rightarrow [EA]$

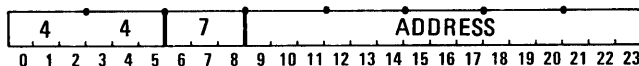
$0 \rightarrow [RA]$

$[[EA \cup 1]]$ shifted in the direction and by the number of bits designated by $[[EA] \cup 1]$, and stored in $[RA]_{0-7}$.

If shift direction bit is 0 and shift count is 0, $[EA \cup 1] - 1 \rightarrow [EA \cup 1]$.

Label SCR Expression

Store Character Right



Description:

SCR stores the leftmost byte of RA into the destination memory location specified by the second operand. The byte location in the destination word is determined by the shift and byte control information specified by the fourth operand. The fourth operand is configured as follows:

[†] Use the value of the term [EA] before execution starts.

Bit 0 Shift direction (0 for right, 1 for left)

Bits 15-17 Byte store control

Bits 18-23 Shift count

The byte control constant is used to store two bytes of the contents of the destination location into RA before the contents of RA are placed back into the destination location. The shift direction and shift count information is used for rotating the leftmost byte of RA into the right position before byte control is applied. Thus, if the fourth operand is 0300, the byte in RA is stored into the leftmost location in the destination word; if the fourth operand is 0510, the byte goes into the middle byte position; and if it is 040000610, the leftmost byte in RA goes into the rightmost location in the destination word.

The first operand points to the appropriate table entry (third and fourth operands); when the instruction is executed the first operand is updated for the next execution by having the third operand copied to the first operand. If the shift direction bit is 1 (rightmost byte), one is added to the second operand to specify the next word in the string for processing. RB is used for storage of intermediate results, and [RA] will be the word as stored back into memory. The first operand must start with SR0T for SCR and with SL2T for SCL when starting on a word boundary.

Note that the Store Right and Store Left tables contain sync bits, which are bit zero of the third operand. These bits are provided for the convenience of the software: the bit is 1 if the beginning of a word has not been reached: thus, for SCR the bit is zero only for the leftmost byte; for SCL it is zero for the rightmost byte. The use of this bit is shown in the first example above at location LPO: the bit is tested and the parallel move loop is entered if it is zero; i.e., if a word boundary has been reached in the destination string.

Condition Codes:



Execution Equation:

$[[EA]] \rightarrow [EA]$

[RA] shifted in the direction and by the number of bits designated by $[[EA] \cup 1]$

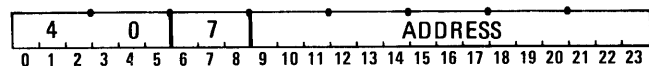
$[[EA \cup 1]] \rightarrow [RA]$, selected bytes

$[RA] \rightarrow [[EA \cup 1]]$

If shift direction bit is 1, $[EA \cup 1] + 1 \rightarrow [EA \cup 1]$

Label SCL Expression

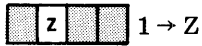
Store Character Left



Description:

SCL operates the same as SCR except that the second operand is decremented by one if the shift count is zero; i.e., if the left boundary of a word has been reached. Also, as noted under SCR, the sync bit is zero for the rightmost byte only, which is convenient for testing when the direction of movement within the string is right to left.

Condition Codes:



Execution Equation:

$$[[EA]] \rightarrow [EA]$$

[RA] shifted in the direction and by the number of bits designated by $[[EA] \cup 1]$

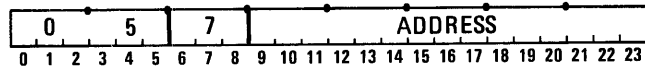
$$[[EA \cup 1]] \rightarrow [RA], \text{ selected bytes}$$

$$[RA] \rightarrow [[EA \cup 1]]$$

If shift count is 0, $[EA \cup 1] - 1 \rightarrow [EA \cup 1]$

Label LPR Expression

Load Parallel Right



Description:

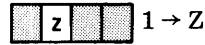
LPR fetches three consecutive bytes from memory and assembles them as a word into RA. The location from which the data is fetched is specified by the second operand (and the second operand plus 1 if required). Which bytes come from which word is specified by the information contained in the fourth operand:

Bit 0	Shift direction (0 for left, 1 for right)
Bits 15-17	Byte store control
Bits 18-23	Shift count

The byte control information determines whether 0, 1, or 2 bytes come from the second data word; the shift direction and shift count information determine what the final alignment of the word will be, suitable for storage into a destination location. Thus, the first word is fetched into RA and then, under byte control, 0, 1, or 2 bytes of the second word are masked into the first word, and the result is rotated if required. If the fourth operand is 0, no bytes from the second word are masked into the first word and the result is not rotated; if the fourth operand is 0410, the left byte of the second word is masked into RA and the word is rotated left 8 to put it on the proper alignment for storing; if the fourth operand is 04000610, the left and middle bytes of the second word are masked into RA and

the result is shifted right 8. RB is used for storage of intermediate results. After execution, one will be added to the contents of the second operand so that it will point to the next word to be operated on; the first operand is *not* changed, as is done for the character instructions.

Condition Codes:



Execution Equation:

$$[[EA \cup 1]] \rightarrow [RA]$$

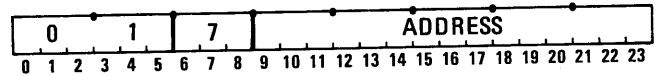
$$[[EA \cup 1] + 1] \rightarrow [RA], \text{ selected bytes}$$

$$[EA \cup 1] + 1 \rightarrow [EA \cup 1]$$

[RA] shifted in the direction and by the number of bits designated by $[[EA] \cup 1]$.

Label LPL Expression

Load Parallel Left



Description:

LPL operates similarly to LPR except that the direction of operation is to the left instead of right. A word is assembled into RA using the data pointed to by the second operand and the next lower-numbered memory location. Which bytes come from which words are determined by the fourth operand:

Bit 0	Shift bit (0 for shift, 1 for no shift)
Bits 15-17	Byte store control
Bits 18-23	Shift Count

First RA is loaded from the location indicated by the second operand, then RA is loaded from the next lower location using byte control. At this point, one is subtracted from the second operand, so that it will point to the next data word. The contents of RA are then rotated as follows: if the shift bit is 1, there is no shift. If the shift count is zero, RA is shifted left 1 byte; if it is non-zero, RA is shifted right one byte. Thus, if the fourth operand is 04000000, the first word remains in RA unchanged; if the fourth operand is 0110, the rightmost byte is taken from the second word and the result is shifted right one byte; if the fourth operand is 0300, the middle and right bytes are taken from the second word and the result is rotated left 8. RB is used for storage of intermediate results. The first operand is not changed.

Condition Codes:



Execution Equation:

$$[[EA \cup 1]] \rightarrow [RA]$$

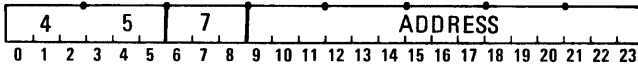
$$[[EA \cup 1] - 1] \rightarrow [RA], \text{ selected bytes}$$

$$[EA \cup 1] - 1 \rightarrow [EA \cup 1]$$

[RA] shifted if sign bit is 0 (shifted left 1 byte if shift count is zero, otherwise right 1 byte).

Label SPR Expression

Store Parallel Right



Description:

The contents of RA are stored in the location specified by the contents of EA. The contents of EA are then incremented by one. After execution, RB contains the updated [EA]. SPR is similar to executing STA* PTR, INR PTR.

In the context of the other Character Move Instructions, SPR assumes that the word has been properly aligned using LPR. The second operand is the [EA] for this instruction; as seen in the first example above, the instruction is written SPR DEST+1. Thus, the contents of RA are placed into the location specified in the second operand, then one is added to the second operand. Note that only the second operand is used by this instruction.

This instruction has many other uses: for example if [0] = 0 and SPR 0 is executed using the REPEAT and STEP switches, the [RA] will be copied into every memory location.

Condition Codes:



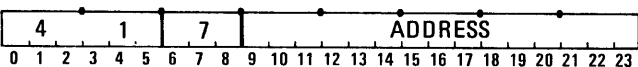
Execution Equation:

$$[RA] \rightarrow [[EA]]$$

$$[EA] + 1 \rightarrow [EA]$$

Label SPL Expression

Store Parallel Left

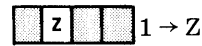


Description:

The contents of RA are stored in the location specified by the contents of EA. The contents of EA are then decremented by one. After execution, RB contains the updated [EA]. Thus, SPL operates exactly the same as

SPR except that the second operand is decremented by one after execution.

Condition Codes:



Execution Equation:

$$[RA] \rightarrow [[EA]]$$

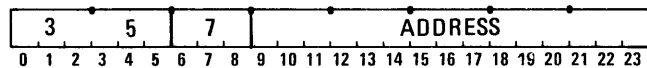
$$[EA] - 1 \rightarrow [EA]$$

Character Translate Instruction

This instruction enables the programmer to look up three characters in a table with a single instruction. A branching feature is provided for special characters.

Label TRT Expression

Translate bytes

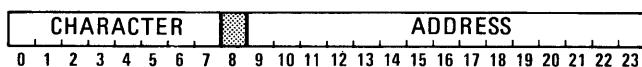


Description:

This instruction is normally used to look up entries corresponding to character codes of various kinds in tables and to record these entries in a variety of ways. The following functions are performed, depending on the specific table entries:

- Replacement of each of the three bytes in a word by separate table entries.
- Addition of the three table entries into a register to create a sum of table entries.
- Branching on special character codes for implementation of error routines, control characters, floating dollar sign, and similar functions.

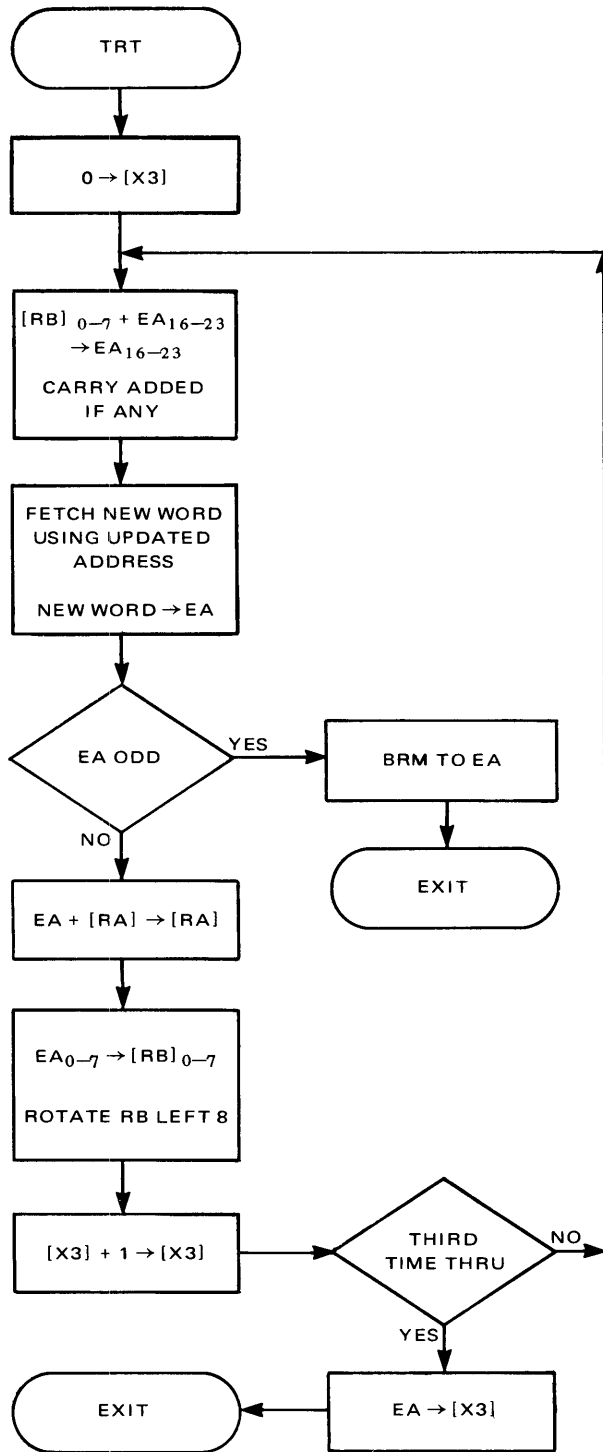
The effective address of the instruction points to the first entry in a table set up by the programmer. The table may contain up to 256 entries, each of which has the following format:



Each table entry must contain both character and address information. The instruction proceeds in the following manner (see flowchart):

- [X3] are set to zero.
- The table is accessed by adding the bits of the left-most character in RB (the byte of data being translated) to a starting address (for the first time through, this is the effective address of the instruction). The resulting sum is used to fetch a table entry.

Flowchart:



- c. If the address field of the table entry is odd, a branch and mark (BRM instruction) is made to that address.
- d. If the address field is even, this address becomes the starting address for the next loop (see step *h*), and the word fetched from memory is added to the contents of RA. RA

is not zeroed in execution of this instruction; therefore the A register may be used to accumulate sums of character and/or address information for more than one execution of the instruction.

e. The leftmost byte of the word fetched from memory is written over the contents of byte 0 in RB; RB is rotated left 8 bits.

f. One is added to the contents of X3.

g. If this is the third time through the fetch loop (see step *h*) of the instruction, the instruction exits with each of the bytes of RB replaced by new information, with each newly fetched word added into RA, and with the last word fetched copied in X3.

h. If this is not the third time through the fetch loop, the instruction returns to step *b*.

The BRM exit from this instruction is used to link to error or exception case routines. Whenever an odd address is encountered and the exception routine is taken, the contents of X3 contain the number of the byte whose table entry caused the branch; testing of X3 provides a ready manner of determining the exceptional character in the RB word. If the normal exit is taken, however, the counter is written over by the last word fetched.

Note that, conventionally, the address part of each non-exceptional table entry is the address of the first table entry (see examples); however, there is no requirement that this be the case. If some different (even) address is used in a table entry, the new address will be added into RA, and the *next* character from RB will be added to the new address. The resulting sum is used to fetch an entry in a different table.

Example 1: Binary quantities, right justified in each of the three bytes of RB (i.e., truncated ASCII characters), are converted to their Excess-Three, Gray equivalents. The results replace the contents of RB, byte-by-byte.

	LDB	N316	
	TRT	TBL1	
	HLT	\$	
	FORCE 0		
TBL1	DCN	00040000+TBL1	0
	DCN	00140000+TBL1	1
	DCN	00160000+TBL1	2
	DCN	00120000+TBL1	3
	DCN	00100000+TBL1	4
	DCN	00300000+TBL1	5
	DCN	00320000+TBL1	6
	DCN	00360000+TBL1	7
	DCN	00340000+TBL1	8
	DCN	00240000+TBL1	9
	DCN	00200000+FIX	DELIMITER
N316	DCN	000600406	
	Before Exec.		After Exec.
	[RB] = 00600406		01203015

Example 2: Using RA, the number of bits in nine digits is computed and stored in the left byte of RA. The digits are given in the binary-quantity-right-justified form from Example 1. Also, a sum of address fields is generated in the right two bytes of RA; when using an extended sum of terms in RA like this example, the programmer should put the table in lower-numbered memory addresses so that the carry from the address field does not enter the byte-sums.

		RCPY	R0, RA, 7
		LDB	N316
		TRT	TBL2
		LDB	N279
		TRT	TBL2
		LDB	N805
		TRT	TBL2
		HLT	\$
		FORCE	0
00146	00000146	TBL2	DCN 0+TBL2
00147	00200146		DCN 000200000+TBL2
00150	00200146		DCN 000200000+TBL2
00151	00400146		DCN 000400000+TBL2
00152	00200146		DCN 000200000+TBL2
00153	00400146		DCN 000400000+TBL2
00154	00400146		DCN 000400000+TBL2
00155	00600146		DCN 000600000+TBL2
00156	00200146		DCN 000200000+TBL2
00157	00400146		DCN 000400000+TBL2
00160	00600406	N316	DCN 000600406
00161	00403411	N279	DCN 000403411
00162	02000005	N805	DCN 002000005

Before Exec.	After Exec.
[RA] = 00000000	03401626

Example 3: Using TBL1 from Example 1, a delimiter (non-convertible to a valid Excess-Three, Gray Character) is sensed, converted to a zero, and counted into a memory counter. X3 is used as a counter of bytes in this example; if the delimiter occurred in byte 0 or byte 1, the exception routine would have to restore neglected characters from the rightward bytes of RB.

	LDB	N88DEL
	TRT	TBL1
	HLT	\$
	FORCE	1
FIX	BSS	1
	RADD	X3, RP, 7
	BRA	BYTE0
	BRA	BYTE1
	BRA	BYTE2
BYTE2	RCL	RB, RB, 7, 8
	RCPY	R0, RB, 1
	INR	COUNTR
	BRR	FIX
	HLT	\$

NUMBR	PZE	0
COUNTR	DCN	-10
N88DEL	DCN	002004012
BYTE0	RCL	RB, X1, 4, 8
	.	
	.	
BYTE1	RCL	RB, X1, 4, 8
		BYTE1 SUBRTN

Before Exec.	After Exec.
[RB] = 02004012	02407000
[COUNTR] = 77777766	77777767
[X3] = Irrelevant	00000002

LIST PROCESSING INSTRUCTIONS

The list processing instructions provide a hardware means of processing queues or stacks of words (24 bits) or characters (bytes or 8 bits). Queues and stacks are defined conventionally: a *queue* is a list where the *first* item entered is the first item considered (first in, first out or FIFO), and a *stack* is a list where the *last* item entered is the first considered, (last in, first out or LIFO). A waiting line at a theater is an example of a queue: the first person arriving gets the first ticket. The stack in an input basket is a typical stack: the last item entered will get first treatment.

List processing within the computer involves tying data of a character or more in length to an address, where the address is that of the next item in the list. The address part of the 24-bit computer word is fifteen bits in length, so that an eight-bit character can be attached to an address. This is the philosophy employed in the IN, UP, and DOWN instructions which, respectively, enter a character into a queue, fetch the character thus stored, or enter the character into a stack. In other applications — such as the implementation of Polish notation — it is convenient to stack words (such as instruction words) in memory. PUSH and POP instructions operate on full words of memory: the PUSH instruction stores a word into a stack if room is available in the stack; the POP instruction similarly retrieves a word at the top of the stack. Both of these instructions operate within the constraints of a stack whose limits are defined by the programmer. Further details of the operations are included within the discussions of each list processing instruction.

The RA and RB registers are used for processing of the information to be stored or fetched. No condition codes are affected by any of these instructions.

Whole Word Stack Instructions

These instructions share a three-word group in memory, defined as follows:

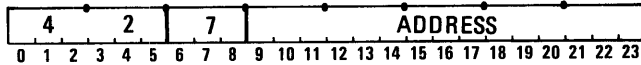
FORCE 0
 PNT0 PZE BTM - 1
 PNT1 PZE PNTR PNT1 is working EA
 PNT2 PZE TOP + 1

BTM and TOP are programmer-defined limits of a block in memory within which the PUSH and POP instructions operate. Each word stored in this block requires one memory location. PNT1 is the location specified by the effective address of the instruction. The contents of PNT1 (PNTR) are set to BTM - 1 before the first word is stored in the stack.

During instruction execution, [RA] are stored into (for PUSH) or fetched from (for POP) the location specified by PNTR. Also, PNTR is compared with TOP for PUSH or with BTM for POP to determine when the stack is full or empty. Note that to poll the stack the programmer merely executes LDA* PNT1.

Label PUSH Expression

Push stack — store [RA] conditionally



Description:

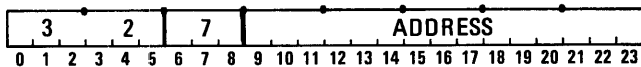
The contents of EA (where EA = PNT1) are fetched, incremented by one, then compared with the contents of PNT2. If equal, the stack is full and the next sequential instruction is fetched (such as a branch to an error routine). If the comparison is not equal, (1) the incremented result is placed in PNT1 and becomes the updated pointer for the next execution; (2) the contents of RA are stored in the location specified by the new [PNT1] where they may be fetched by a later POP instruction; and (3) the next sequential instruction is skipped.

Execution Equation:

$$\text{If } [EA] + 1 \neq [EA + 1] \text{ then } [RP] + 1 \rightarrow [RP], [EA] + 1 \rightarrow [RB], [EA] + 1 \rightarrow [EA], [RA] \rightarrow [[EA]]$$

Label POP Expression

Pop up stack — fetch [[EA]] conditionally.



Description:

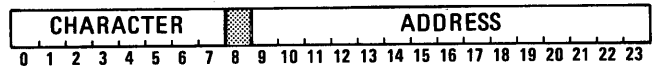
The contents of the location specified by the contents of EA (where EA = PNT1) are fetched and placed in RA. The contents of EA are then compared with the contents of PNT0. If equal, the stack is empty and the next sequential instruction is executed. If the comparison is not equal, the contents of PNT1 are decremented by one and the next sequential instruction is skipped.

Execution Equation:

$$[[EA]] \rightarrow [RA]. \text{ If } [EA] \neq [EA - 1] \text{ then } [RP] + 1 \rightarrow [RP], [EA] - 1 \rightarrow [RB], [EA] - 1 \rightarrow [EA]$$

Character List Processing Instructions

The UP, DOWN, and IN instructions can be used conveniently for saving and retrieving characters in lists. Each item in a list contains a character and an address of the next character in the list as follows:



An address of zero indicates the last character to be removed from a queue or a stack.

These instructions normally share a list of available storage space from which a location may be obtained before adding a character to any stack or queue and to which a location may be returned after fetching a character from any stack or queue. Figure 5-3 illustrates these instructions.

Label UP Expression

Up list



Description:

This instruction fetches a character from the front of a queue or the top of a stack as designated by the pointer in EA and places the character in the left byte of RA and RB. The remaining two bytes of RA are loaded with zeros so that the character may be ORed into the left byte of a word if desired. The pointer in EA is loaded into the address part of RB; this allows the location containing the fetched character to be returned to a stack of unused locations set aside for lists by using the DOWN instruction.

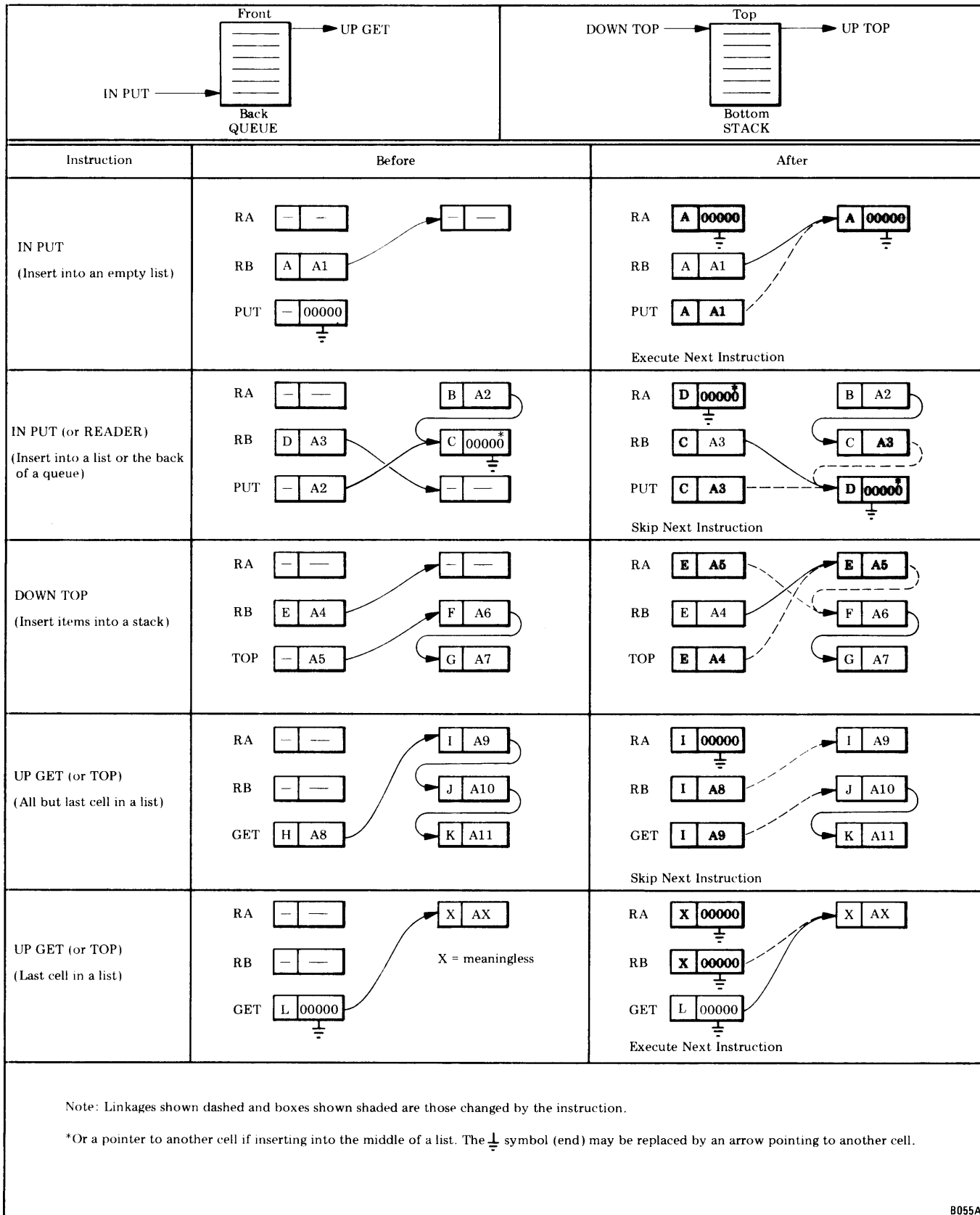
If the address part of the contents of EA is zero (i.e., the list is empty), the next sequential instruction is executed. Otherwise, the contents of the location specified by the contents of EA (the fetched character and pointer to the next character) are stored in the contents of EA and the next sequential instruction is skipped.

The conventional uses of this instruction are: (1) to fetch an empty cell from a stack for use by DOWN or IN; (2) to fetch the character at the top of a stack or front of a queue. See "Examples" following.

Execution Equation:

$$[EA] \rightarrow [RB], 0 \rightarrow [RA], [[EA]]_{0-7} \rightarrow [RA]_{0-7}, [[EA]]_{0-7} \rightarrow [RB]_{0-7}$$

$$\text{If } [EA]_{9-23} \neq 0 \text{ then } [[EA]] \rightarrow [EA], [RP] + 1 \rightarrow [RP]$$

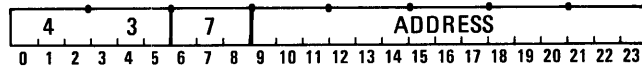


B055A

Figure 5-3. Character List Processing Instructions

Label DOWN Expression

Down list



Description:

This instruction adds the character contained in the left byte of RB to the top of a stack with the address part of RB designating the location the character is stored in. The pointer in EA, which specifies the previous top character of the stack, is loaded into the address part of RA and stored in the address part of the location containing the new top character. The new character is also loaded into the left byte of RA and EA. The pointer to the new character is loaded into the address part of EA.

The conventional uses of this instruction are: (1) to put a character into the top of a stack; (2) to return an empty cell to a stack for future use. See "Examples" following.

Execution Equation:

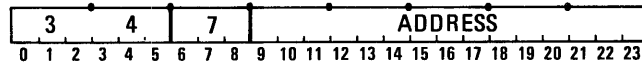
$$[EA] \rightarrow [RA], [RB]_{0-7} \rightarrow [RA]_{0-7}$$

$$[RB] \rightarrow [EA], [EA]_{9-23} \rightarrow [[RB]]_{9-23}$$

$$[RB]_{0-7} \rightarrow [[RB]]_{0-7}$$

Label IN Expression

Insert in list



Description:

This instruction inserts the character contained in the left byte of RB into a list immediately after the character specified by the pointer in EA. The instruction is normally used to add a character to the end of a queue. The instruction is executed as follows: The pointer in the location preceding the inserted character is loaded into the address part of RA. The new character is then loaded into the left byte of RA.

If the address part of [EA] is zero (i.e., the list is empty), the zero address is loaded into RA and stored in the address part of the location specified by the pointer in RB. The character in the left byte of RB is also stored in this location, which is the first location in a new list. The contents of RB are also stored in the location specified by EA.

If the address part of [EA] is not zero, the character specified by the pointer in EA is loaded into the left byte of RB and EA. The pointer in RB is stored in EA and the location containing the character preceding the inserted character; this location is specified by the pointer in EA. The pointer originally in this location is stored in the location specified by the pointer in RB. The new character,

in the left byte of RB, is then stored in this location and the next sequential instruction is skipped.

Execution Equation:

$$[[EA]] \rightarrow [RA], [RB]_{0-7} \rightarrow [RA]_{0-7}$$

$$\text{If } [EA]_{9-23} = 0 \text{ then } 0 \rightarrow [RA]_{9-23}, [RB] \rightarrow [EA],$$

$$0 \rightarrow [[RB]]_{9-23}, [RB]_{0-7} \rightarrow [[RB]]_{0-7}$$

$$\text{If } [EA]_{9-23} \neq 0 \text{ then } [[EA]]_{0-7} \rightarrow [RB]_{0-7},$$

$$[RB]_{9-23} \dagger \rightarrow [[EA]]_{9-23}, [[EA]]_{0-7} \rightarrow$$

$$[EA]_{0-7}, [RB]_{9-23} \dagger \rightarrow [EA]_{9-23}$$

$$[[EA]]_{9-23} \dagger \rightarrow [[RB]]_{9-23}, [RB]_{0-7} \dagger \rightarrow$$

$$[[RB]]_{0-7}, [RP] + 1 \rightarrow [RP]$$

Examples:

By convention, the character instructions work with a block of storage called FREE in the assembly language, and use three pointers: TOP, the pointer to the latest entry in a stack; GET, the pointer to the earliest remaining entry in a queue; and PUT, the pointer to the latest entry in a queue. At the beginning of a queuing routine, GET will be zero and PUT will point to GET, which means that the queue is empty. Note that the UP instruction can detect an empty queue after the last character is taken from the queue. However, the pointers to the two ends of the queue are unsynchronized and a new character can be lost when in an interrupt-driven environment. Therefore, the software must check for the zero condition. Note that the interrupts to this level must be disabled during the checking period.

Example 1: An LCR instruction is used to obtain a character to be queued. UP FREE obtains a storage location in which to save the character. IN PUT puts the character at the end of the waiting line. The routine at CONSM1 might be an output routine.

START	LD3	M3	
QUEUE	LCR	SOURCE	Get character
	RCPY	RA,X1	
	UP	FREE	Get cell
	HLT	\$	Won't happen
	RCPY	X1,RB,4	
	IN	PUT	Put character
	NOP		Won't happen
	BC3	QUEUE	Build loop
FETCH	PID	ALL	No interrupts
	UP	GET	Take character
	BRA	DONE	Done with queue
	DOWN	FREE	Return cell
	LDA	MASK	Is address
	AND	GET	zero
	BNZ	CONSM1	Use character
	LDA	@GET	Fix pointers
	STA	PUT	

† Use the value of the term before execution starts.

CONSM1	PIA	ALL	Routine to use character in RB	RCPY	X1,RB,4	
	.			DOWN	TOP	Put character away
	.			BC3	STACK	Build loop
	.			TAKE	UP	Retrieve character
	BRA	FETCH	Retrieve loop	BRA	DONE	Stack empty
DONE	BSS	0		DOWN	FREE	Put cell back
	.			CONSM3	BSS	0
	.			.		Routine to use character in RB
	.			.		
GET	PZE	0	Pointer to top			
PUT	PZE	GET	Pointer to bottom		BRA	TAKE
MASK	DCN	077777	of queue	DONE	HLT	\$
M3	DCN	-3		TOP	PZE	0
@GET	PZE	GET			END	NEXT
	FORCE	0				
SOURCE	DCN	LR0T	From char. tables			
	DCN	ALPHA	in figure 3-2.			
ALPHA	DCA	@ABCDEF@				
ALL	DCN	0377	Interrupt mask			
FREE	PZE	\$+1	FREE list			
	PZE	\$+1				
	PZE	\$+1				
	.					
	.					
	.					
	PZE	0	End of FREE			
	END	START				

The code is substantially the same, except that IN PUT is replaced by DOWN TOP, and there is no equivalent of the testing and synchronizing logic after UP GET. The execution of this loop will store D, E, and F in a stack, and then fetch F, E, and D from the stack in that order. Note that the characters are fetched in the reverse order from the queue example.

Example 3: An important feature of the IN instruction is the ability to insert a cell into a list while maintaining the integrity of the pointers in the list. In this example, a queue is built after the manner of Example 1, with the letters "LISED" in order. The queue is then read using UP READER (where READER is set equal to the GET pointer at the start) and tested until the desired insertion point is found. The letter T is then inserted (using IN READER) to make LISED into LISTED. Note that after the insertion point is found, it is necessary to backspace one cell (STB READER) so that the pointers will align properly.

FIX	LD3	M5	
BUILD	LCR	WORD	Make a queue
	RCPY	RA,X1	
	UP	FREE	Get cell
	HLT	\$	
	RCPY	X1,RB,4	
	IN	PUT	
	NOB		
	BC3	BUILD	Build loop
	LDA	GET	Start a read
	STA	READER	Pointer
TEST	UP	READER	Read the queue
	NOB		
	CPA	SMASK	Test char. in RA
	BNZ	TEST	Test loop
INSERT	STB	READER	Backstep one
	UP	FREE	Get cell
	NOB		
	LD1	T	Get correction
	RCPY	X1,RB,4	Into RB
	IN	READER	Insert
	NOB		
	BRA	FETCH	Go read queue

Execution of this loop will store A, B, and C in a queue, and then fetch A, B, and C from the queue in that order. Note that after the execution of any such loop, the order of pointers within the FREE list will be changed — however, the integrity of the list will be preserved in that each FREE location will point to another FREE location, except the last, which points to zero.

A similar method is used to build and read a stack, except that the stack does not require the auxiliary software required to keep two pointers in synchronization. The basic instruction sequences for the stack and queue are contrasted in the following table:

Instruction Use	For Queue	For Stack
Get Free cell	UP FREE	UP FREE
Put Character	IN PUT	DOWN TOP
Get Character	UP GET	UP TOP
Return cell to Free List	DOWN FREE	DOWN FREE

Example 2: The corresponding program for processing a stack, with labels consistent with the queue example, is as follows:

NEXT	LD3	M3	
STACK	LCR	SOURCE	From example 1
	RCPY	RA,X1	
	UP	FREE	Get cell
	HLT	\$	No more FREE

M5	DCN	-5	
	FORCE	0	
WORD	DCN	LR0T	From character tables
	DCN	WORD1	
WORD1	DCA	@LISED@	Misspelled
SMASK	DCA	@S@0	Right 2 bytes of RA will be zero
*			
T	DCA	@T@0	
READER	PZE	GET	
	END	FIX	

These examples illustrate use of the three instructions for processing lists of characters. Use of a free list constructed in two word blocks (FREE PZE \$ + 2), for example, affords the possibility of pairing a data word with each address word.

DECIMAL OPTION INSTRUCTIONS

The decimal option provides instructions for high speed manipulation of words and characters under program control. Decimal numbers may be added or subtracted, character strings may be compared, numbers may be changed, and byte strings may be moved rapidly using these instructions. The instructions are as follows:

DADD	Decimal Add (ASCII)
DSUB	Decimal Subtract (ASCII)
MVCL	Move Characters Left
MVCR	Move Characters Right
CPL	Compare Logical (bit-by-bit)
CPN	Compare Numeric (ASCII)

These instructions operate from a memory Source Location to a Destination Location. Before execution, the Source Address must appear in X2 and the Destination Address in X3. During execution, RB, X2, and X3 are used for generating addresses; their contents will be meaningless after execution.

Note that the MVCR and CPL instructions operate on data from left to right, whereas MVCL, DADD, DSUB, and CPN operate from right to left. MVCL, MVCR, and CPL do not recognize sign information; they treat all bits alike. In the DADD, CPN, and DSUB instructions, however, the least significant (rightmost) byte contains sign information in its zone field, encoded as follows:

2nd, 3rd & 4th Bits	Sign
011	+
100	+
101	-

Thus, the least significant digit for negative numbers is represented by the letters P through Y for the digits 0 through 9. Note that the parity bit (first bit of each byte) is never changed by any of the six decimal instructions.

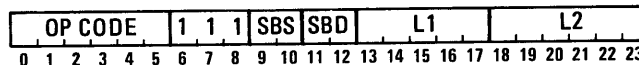
The formats for these instructions differ from those for other instructions. The MVCL, MVCR, and CPL instructions use the following format:



SBS and SBD are the Starting Byte of the Source and Destination locations, respectively; i.e., the first byte to be operated upon when execution begins. SBS and SBD are encoded as follows:

SBS or SBD Field	Starting Byte
00	0
01	1
10	2
11	Illegal

L is the length in bytes, minus 1, of the block to be operated upon. The maximum number of bytes is 256. DADD, DSUB, and CPN have the following format:



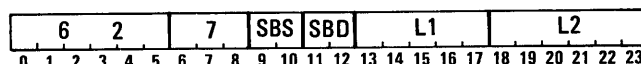
SBS and SBD are the same as for MVCL, MVCR, and CPL.

L2 is the length of the source quantity, in bytes, minus 1. The maximum number of bytes is 64.

L1 is the difference between the lengths of the source and destination quantities, in bytes. The maximum difference is 31. Note that the source may not be longer than the destination.

Label DADD SBS,SBD,L2,L1

Decimal Add



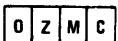
Description:

The contents of X2 and the SBS are combined to select the least significant byte (highest-numbered address location) of the source quantity. The contents of X3 and SBD are combined to select the least significant byte of the destination quantity. The zone bits of these bytes form the sign of the two quantities, encoded as shown above. These two bytes are added as ASCII numbers, and the result is placed back in the destination location. The next sequential characters are fetched and added, and the addition continues from right to left with the carries propagated in the normal decimal manner. If the operands are of unequal length (i.e., L1 is not zero), then the computer supplies ASCII zeros for the high order bits of the source operand to complete the add.

Addition take place algebraically with regard to sign. If the intermediate result in the Destination location is a negative ten's complement number, it is recomplemented and a negative sign is attached. The recomplementation operation is the only operation that will change the sign zone bits.

The condition codes are set to reflect the results of the operation. If the result is negative, the minus condition code will be set. If the result is zero, the zero condition code will be set; a minus zero is possible only if there is an overflow. If the result is recomplemented, carry will be set and overflow reset; note that recomplementation is automatic. If there is an overflow (e.g., if the destination area was one byte too short and a 1₁₀ needs to be appended to the front of the destination quantity) the overflow and carry condition codes will be set. If not, overflow will be reset, unlike the case with the non-decimal hardware, where the overflow condition code is not reset unless it is tested. The software must make adjustments for the overflow condition.

Condition Code:

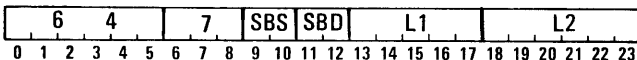


Execution Equation:

[S Memory Block] + [D Memory Block] → [D Memory Block], ASCII.

Label DSUB SBS,SBD,L2,L1

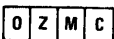
Decimal Subtraction



Description:

This instruction operates exactly like DADD except that the source quantity is subtracted from the destination quantity. The result replaces the destination location, with the sign locations set appropriately. The condition codes are set just as they are for DADD.

Condition Code:



Execution Equation:

[D Memory Block] - [S Memory Block] → [D Memory Block], ASCII.

Label CPN SBS,SBD,L2,L1

Compare Block Numeric (ASCII)



Description:

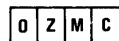
This instruction performs a numeric comparison operation on two blocks of memory, and sets the condition codes accordingly. The comparison is effected by subtracting the source memory block from the destination block, but the result is not stored. For the condition code convention, see DADD, above. Note that CPN treats a minus zero as equal to a plus zero. See Table 5-1 for the result of a numeric comparison.

Table 5-1. Numeric Comparison Results

Condition	Condition Code Settings†			
	O	Z	M	C
[D] = [S]	0	1	0	X
[D] > [S]	0 1	0 X	0 0	X 1
[D] < [S]	0 1	0 X	1 1	X 1

† X indicates don't care

Condition Code:



Execution Equation:

[D Memory Block] : [S Memory Block], ASCII, set condition codes.

Label CPL SBS,SBD,L

Compare Logically, bit-by-bit



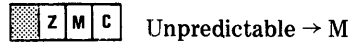
Description:

The contents of X2 and the SBS are combined to select the leftmost byte (lowest numbered location) of the source character string. The contents of X3 and SBD are combined to select the leftmost byte of the destination character string. The two strings must be of equal length.

The characters of the destination string are compared to the characters of the source string on a bit-by-bit basis, one character at a time, left to right. If no difference is found, the strings are equal, and the instruction ends with the zero CC set to 1. If a difference is found, the instruction ends immediately with the zero CC set to 0. If the source string is greater logically (1 bit found in the source string at the first difference), the carry CC is set to 1; otherwise the carry CC is set to 0. The minus CC is unpredictable.

Note that collating sequences generated using this instruction will be altered if parity bits are present in the ASCII characters that are tested. In general, it is recommended that parity bits be masked off of data before processing in the computer.

Condition Codes:



Execution Equation:

[D Memory Block] : [S Memory Block], bit-by-bit, set condition codes

Label MVCR SBS,SBD,L

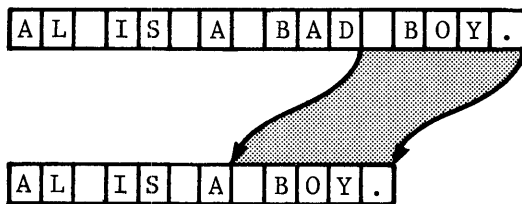
Move Characters Right



Description:

The contents of X2 and SBS are combined to select the leftmost byte of the source block to be transferred, and the contents of X3 and SBD are combined to select the first (leftmost) byte of the destination to which transfer is to be made. The source byte string and the destination memory area must be the same length. The bytes of the source replace the bytes of the destination, a byte at a time from left to right (from lower numbered address to higher numbered address). This instruction is intended for delete editing not insert editing in the case of overlapping fields.

Example:



Condition Code:

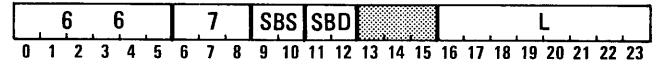


Execution Equation:

[S Memory Area] → [D Memory Area], left to right

Label MVCL SBS,SBD,L

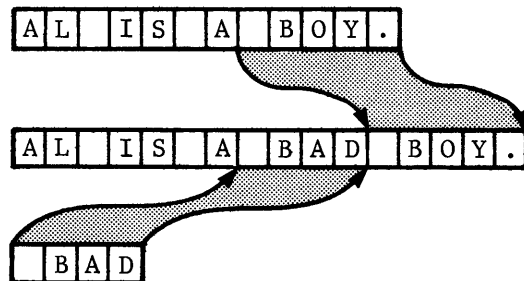
Move Characters Left



Description:

This instruction operates identically to MVCR except that the move is from right to left, byte by byte. Thus, the contents of X2 and SBS are combined to select the rightmost byte of the source, and the contents of X3 and SBD are combined to select the rightmost byte of the destination. The move then proceeds from right to left. Note that this instruction is intended to be used for insert editing, where overlapping fields are involved; MVCR is convenient for delete editing not insert editing.

Example:



Condition Code:



Execution Equation:

[S Memory Area] → [D Memory Area], right to left.

Section 6

Input/Output System and Instructions

The input/output system allows the CPU to communicate with peripheral units, other devices, and the control panel under program control. All input/output transfers involved in communicating information are *performed* directly under program control using the instructions described in this section. Most input/output transfers are *initiated* using the interrupt system described in Section 7.

System characteristics allow:

- Block transfers of words between a device and memory under direct CPU control at rates up to 375,000 bytes/second and under control of the interrupt system at rates up to 39,000 bytes/second.
- Single word transfer of control and status information to and from devices.
- Recognition of up to 512 separate device addresses grouped into eight channels of 64 devices each.
- Concurrent operation of multiple channels, and multiple devices within channels, up to system bandwidth limitations.

- Automatic output of information stored in dedicated storage locations to video devices without need of program intervention except for changing the information displayed.

The input/output system includes: peripheral unit I/O, external command and external sense I/O, and console keys input.

PERIPHERAL UNIT I/O

The following paragraphs describe the organization of the peripheral unit I/O, device addresses, peripheral units, I/O instructions, and the execution of I/O instructions.

Organization

Figure 6-1 illustrates the structure of the peripheral unit I/O. The main components are the I/O interface logic, the I/O channels, the peripheral controllers, and the I/O buses.

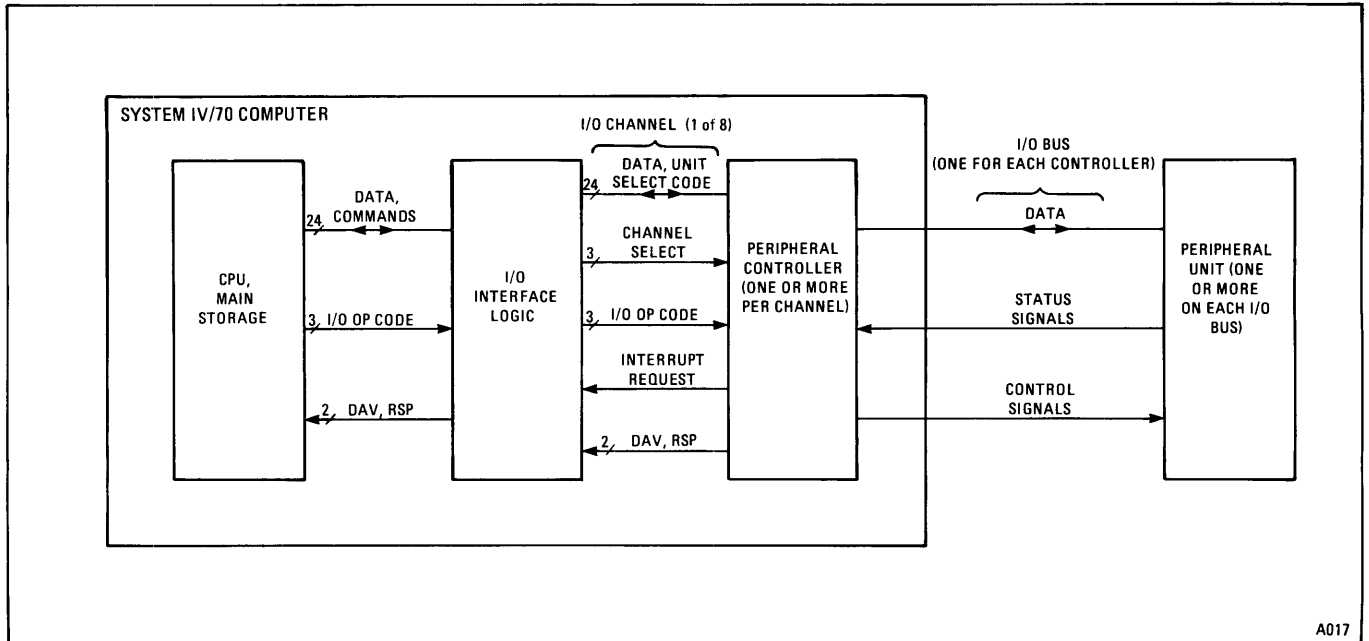


Figure 6-1. Peripheral Unit I/O Structure

I/O INTERFACE LOGIC

The I/O interface logic interfaces the peripheral controllers with the CPU and main storage by processing all data and control signals used to input and output information.

I/O CHANNELS

Eight I/O channels are used to connect peripheral controllers to the I/O interface logic. One or more controllers may be connected to each channel depending on the type of controller and the system configuration. Each channel contains the following lines:

- Eight or 24 bidirectional data lines for transferring bytes or words to and from peripheral units. Eight of the data lines are used for unit selection and unit identification at the appropriate time.
- Three I/O op code lines for selecting I/O operations: select external device, acknowledge interrupt, input data, and output data.
- Three channel select lines.
- An interrupt request line with which the peripheral controller signals the I/O interface logic when the device needs to be serviced by interrupting the program being processed.
- An RSP line with which the controller signals the CPU that it is responding to a "select external device" signal.
- A DAV signal with which the controller signals the CPU that it is ready to send or has received data.

PERIPHERAL CONTROLLER

Peripheral controllers electrically and functionally match each peripheral unit to an I/O channel. Each controller may control one or more units depending on the type of unit. A controller is contained on one or more printed circuit cards housed in either the Processing Unit or the expansion cabinet. The controllers contain logic for channel and unit address recognition as well as control logic. Also, buffering of eight or 24 bits of data is provided so that the CPU may service the device a byte or word at a time. Each controller has the ability to generate an interrupt request, to receive control information (if any), to respond to requests for status information, and to input and/or output the data as required.

I/O BUS

The I/O bus connected to each controller transfers status signals, control signals, and data between the controller and the corresponding peripheral units. The I/O bus is connected to the peripheral units in "daisy chain" fashion allowing additional units to be added by simply extending the bus.

Device Address

Each peripheral unit attached to the computer has a device

address. This address includes a 3-bit channel number and a 6-bit unit number. When an input/output operation takes place, this device address is sent from the CPU to alert the particular peripheral controller that a transfer is desired.

Each of the eight channel numbers is assigned to a unique interrupt level. This assignment simplifies the decoding in the peripheral controllers since the channel address and the interrupt level have the same number.

Peripheral Units

The I/O system handles various-speed peripheral units in a manner most economical for the particular unit. Peripherals fall into three groups:

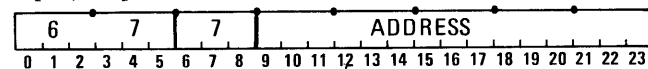
- *"Lock-up" Devices.* These high speed devices require the CPU to lock-up to the device being serviced due to the high data rates involved. This group includes devices such as disc files and magnetic tape units.
- *Synchronous Devices.* These devices include slow-to-medium speed devices which need not be serviced as frequently as lock-up devices, but do require servicing within a fixed time interval. For example, once a card read is initiated, there is a maximum permissible time that the controller can wait before receiving data. If the time period is exceeded, the data in that column is lost and the card must be reread.
- *Asynchronous Devices.* These devices, by their hardware nature, may be serviced at any time. This group includes devices such as printers and data terminals.

I/O Instructions

Three instructions (IO, IOB, and BOOT) are used to input and output information using peripheral units. These instructions use a select (CUT) word for selecting the channel, unit, and type of information and a buffer-address word for designating the main storage location where information is stored. Figure 6-2 shows the format of the select word and the buffer address word and describes the significance of each part. See the "Peripheral Unit Programming Manual", document SIV/70-40-1, for detailed programming information on the various peripheral devices.

Label IO Expression

Input/Output Words



Description:

Most input and output in the system is accomplished using this instruction. The effective address, which must be even, contains the select word; the effective address ORed with 1 contains the buffer address word, which is incremented by one for each data, control, or status transfer. See Figure 6-2. It is normal practice to decrement (using the DEC instruction) the buffer address word after a status or control IO instruction.

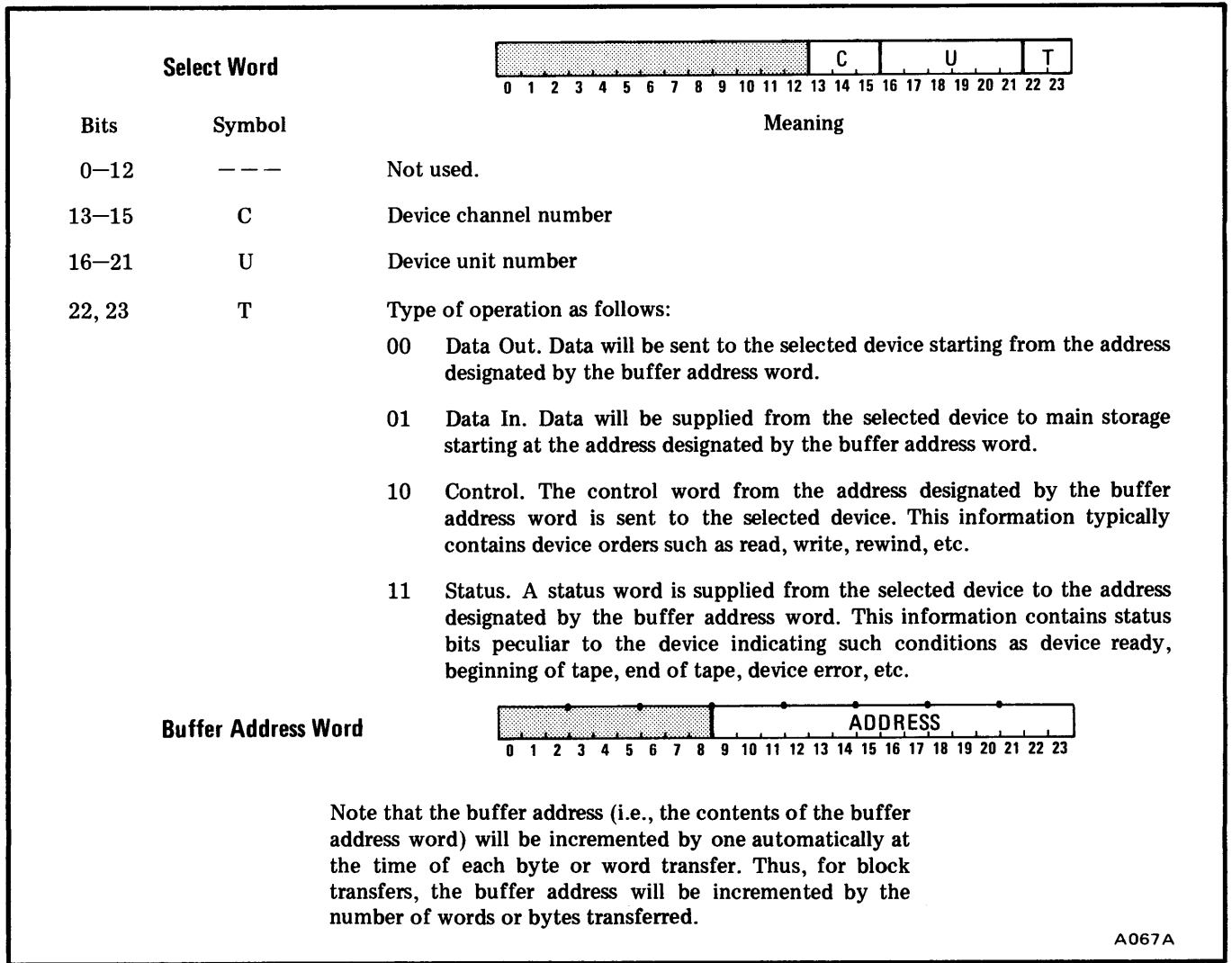
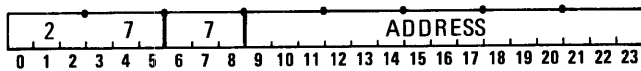


Figure 6-2. Select Word and Buffer Address Word Formats

Label IOB Expression

Input/Output Bytes



Description:

Inputs bytes and packs them into a word or unpacks bytes from a word and outputs them. The effective address, which must be even, contains the select word; the effective address ORed with 1 contains the buffer address word. See Figure 6-2. On input, IOB assembles three 8-bit characters from a peripheral unit and stores them at the location designated by the buffer address. On output, IOB disassembles three characters obtained from the buffer address location and sends them to a peripheral unit. The B register is used for the assembly and disassembly of a word; the A register must contain a byte control of 1 and a shift count of 7 formatted as follows: [RA] = 00000107₈. The contents of the B register are meaningless after execution.

Condition Codes:



Label BOOT S,D

Bootstrap Load



Description:

Used to initialize the computer whenever power is turned on by inputting a loader program and/or assembler program into main storage. The loader program is used to load any programs that have been assembled by the assembler or compiled by a compiler. The destination register may be any register which contains the select word shown in Figure 6-2. The source register must be R0 for inputting words

from the device and must be RA for inputting 8 bit characters and packing them into a word. In this case RA contains 40000107 which represents pack, byte control, and shift count information.

RP is used as a counter in the loading of the bootstrap program; it is initially set to 1 at the start of the BOOT instruction and the first word of the bootstrap program is then loaded into location 1 of main storage. Each time a word is loaded by the BOOT instruction, 1 is added to [RP], and the next word is stored using the address thus generated. The loading proceeds until the bootstrap device stops sending (the procedure by which the peripheral unit controller decides transmission is complete is device dependent).

When the device stops sending, the current contents of RP (the location of the last word stored plus one) and the contents of location 1 are swapped and the computer starts operating in normal AUTO mode. This has the same effect as treating the first word read in as a branch instruction; the first instruction actually executed will be fetched from the location pointed at by the first word loaded in the bootstrap procedure. The word thus loaded is conventionally a BRA START, where START is the starting location of the program. The word stored in location 1 will be one more than the count of the number of words loaded by the BOOT instruction; this is also the last location loaded plus 1.

To initiate a bootstrap load on systems with a BOOT switch, set all three DISPLAY SELECT switches down and put the boot word into the 24 console keys. The BOOT instruction goes into the first five octal digits and the select word goes into the last four; note that the data overlaps all but the first bit of the fifth digit and thus the channel address will determine what destination register can be used. Next, ready the peripheral unit, and then press the BOOT switch to start operation.

To initiate a bootstrap load on systems with an INTERRUPT switch, select MANUAL mode, press SYSTEM RESET to put the controller in the bootstrap mode, press STEP to exit from the system reset mode, key in the contents of the source and destination registers, key the BOOT instruction into the TIR, ready the peripheral unit, and then select the AUTO mode to start operation.

Condition Codes:



Execution of I/O Instructions

All I/O peripheral unit I/O instructions are executed using the responsive (hand-shaking) method of information transfer. When outputting information, the word (or byte) is presented to the unit until it indicates that it received the word (DAV signal true). The CPU presents additional words to the unit until the unit indicates that it has received the required data or its limit of data (RSP signal false). When in-

putting information, the CPU waits until the unit indicates that it is ready to send a word (DAV signal true) and then gates the word into the CPU. The CPU remains connected to the unit accepting additional words in the same manner until the unit indicates that all the data has been sent (RSP signal false).

The following events take place during the execution of an I/O instruction. The CPU sends the select word (see Figure 4-2) to the peripheral controller along with a select-external-device (SE) command. This has the effect of: (1) selecting the appropriate peripheral controller and peripheral unit (channel and unit number); (2) signalling that a data transfer is being initiated (SE); and (3) specifying the type of operation (type field). Note that sending an address that is not assigned to any controller is an error that will hang the computer in a loop that can only be cleared by activating SYSTEM RESET. The selected controller answers by setting the response (RSP) signal true as soon as it receives the SE command. The CPU then drops the select word and the SE command, proceeds to swap [RP] and [buffer address location], and then determines from the select word whether the transmission is an input to or an output from main storage.

INPUT TRANSMISSION

If this is an input transfer, the CPU issues an input external (IE) command to the controller and waits for a true data-available (DAV) signal. In response to the IE command, the controller places the word on the data bus and sets the DAV signal true. The CPU drops the IE command, loads the word into the memory data register (MDR), and stores the word at the location specified by the buffer address in the program counter.

If bytes instead of words are being loaded and packing is specified (IOB instruction or a pack BOOT instruction), two additional bytes are inputted using the IE command. All three bytes are assembled in RB, loaded into the MDR, and stored in memory. If a multiword transfer is taking place (lock-up devices), the controller will supply the additional words and the CPU will increment the program counter appropriately to provide new buffer addresses.

When the controller is finished with a transfer, it sets the DAV and RSP signals false. This is true for one-word or multiword transfers. The CPU adds one to the buffer address in the program counter† and swaps [RP] and [buffer address location]. This provides a new buffer address for the next input operation and restores the address of the next instruction to the program counter for execution. If the I/O instruction is the first instruction (excluding IOID) executed as the result of an interrupt, a debreak signal is issued to the I/O interface logic before exiting from the I/O instruction; this clears the interrupt just serviced.

† Note that a carry may develop into the most significant bits of the buffer address word and therefore these bits may be meaningless.

OUTPUT TRANSMISSION

If this is an output transfer, the CPU sends the contents of the buffer address specified by the program counter to the controller along with an output-external (OE) command. When the controller accepts the word, it sets the DAV signal true. The CPU then updates RP by one, fetches the next word from the address thus generated, and sends it to the controller. If a multiword transfer is taking place (lock-up device), the controller will accept the word and the above process will be repeated. Otherwise, the controller will set RSP false and not accept the word. The buffer address will now be one greater than the last word accepted; i.e., it will be the next address from which output is required. The CPU then swaps [RP] and [buffer address location] to keep the updated buffer address and to restore the address of the next instruction to the program counter. If the I/O instruction is the first instruction (excluding IOID) executed as the result of an interrupt, a debreak signal is issued to the I/O interface logic to clear the interrupt just serviced.

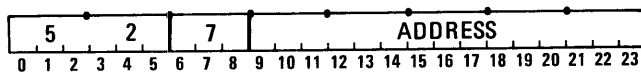
During unpacking (IOB instruction) the following events take place in addition to those described above. When the contents of the buffer address are fetched, the word is disassembled into three bytes using RB and sent to the controller one byte at a time. Note that this instruction destroys [RB]. The controller indicates acceptance of each byte by setting the DAV signal true. The CPU updates RP by one after the first byte has been accepted† and sends the first byte of the next word to the controller after all bytes of the present word have been accepted.

EXTERNAL COMMAND AND EXTERNAL SENSE I/O

The external command and external sense instructions operate completely independently of the peripheral unit I/O channels. The external command instruction allows the user to simultaneously generate up to four control signals for general purpose external or internal use. The external sense instruction allows the user to simultaneously sense up to four different conditions from external or internal sources.

Label EXCT Expression

External Command



Description:

The four low order bits of the contents of the effective address ([EA]₂₀₋₂₃) are placed on the four external command lines (EXC) of the computer. The presence of a bit in any position will generate a 4-microsecond output pulse on the corresponding line.

These lines are used to control the memory parity circuits. Control codes are 14₈ = enable parity, 15₈ = disable parity,

† Note that a carry may develop into the most significant bits of the buffer address word and therefore these bits may be meaningless.

16₈ = select odd, 17₈ = select even.

Condition Codes:

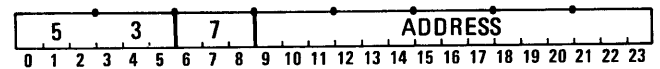
None.

Execution Equation:

$$[EA]_{20-23} \rightarrow [EXC]_{0-3}$$

Label EXSN Expression

External Sense



Description:

The four low order bits of the contents of the effective address ([EA]₂₀₋₂₃) are compared with the computer's four external sense (EXS) lines. If any of the corresponding lines and bits are both ones, the computer skips the next instruction.†

Condition Codes:

None.

Execution Equation:

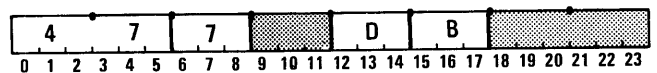
$$\text{If } ([EA]_{20} \cap EXS_0) \cup ([EA]_{21} \cap EXS_1) \cup ([EA]_{22} \cap EXS_2) \cup ([EA]_{23} \cap EXS_3) = 1, \\ [RP] + 1 \rightarrow [RP]$$

CONSOLE KEYS INPUT

This instruction allows the setting of the console keys to be stored in a register.

Label ECS D,B

Enter Console Switches



Description:

The contents of the console keys replace the designated characters of the destination register. If no byte control is given, the assembler will furnish 7 (all bytes).

Example:

The instruction is ECS RA,1.

	Before Exec.	After Exec.
[Keys]	= 07007707	07007707
[RA]	= 55555555	55555707

Condition Codes:

None.

Execution Equation:

$$[Keys] \rightarrow [D], \text{ Selected Bytes}$$

† This instruction is available only on the 7002 Processing Unit.

Section 7

Interrupt System and Instructions

Interrupts allow external events (or certain internal software conditions) to alter the Processing Unit's currently programmed course of actions. When interrupted, the processor will perform an I/O transfer or some other interrupt "service", then return to the programmed course as if no disturbance had occurred. An interrupt is performed in response to an interrupt-request signal to the processor that may originate at a peripheral controller or from the processor itself. Interrupt requests may be generated for a variety of reasons, typical of which are:

- An input or output data transfer is required or possible.
- A significant change of status has occurred in a peripheral controller (e.g., printer out of paper; card reader needs a pick command).
- A program at a higher interrupt level has requested an interrupt at a lower level (e.g., using the EXCT instruction).
- Achieving a zero count when using the INR instruction in an interrupt location.

When an interrupt request is received, the processor's internal logic examines the request and initiates a new course of action (interrupt) at the appropriate point in its cycle of operation. The current status of affairs in the program is kept (either undisturbed or restorable as explained below), and a hardware transfer (an execute, not a branch) is forced to a memory location assigned to the device requesting the interrupt. The instruction in this location is fetched and executed to accomplish the course of action for which the interrupt was requested. At the completion of this instruction, or the series of instructions it initiates, a signal will be issued to cause the interrupted program to pick up and continue as if nothing had happened.

PRIORITY INTERRUPT LEVELS

Eight unique levels of interrupts are provided, with true priority nesting within the levels. Each level is assigned a unique memory location that contains an instruction for servicing (or initiating a routine to service) the device or software program initiating the interrupt. Within each of the eight levels, up to 64 device or "unit" addresses may be assigned; the way these are used is explained under "Indirect Interrupt Processing" below.

The highest priority interrupt has the lowest memory location (see Table 7-1 "Dedicated Interrupt Locations"). The levels are fully nested since (a) an interrupt of higher priority can interrupt a lower level, but a lower level cannot interrupt a higher level, and (b) the higher level can be interrupted by a still higher level, up to the maximum eight levels (plus the

background). The eight interrupt levels are identified with the eight I/O channels described in Section 6 under "Device Address"; each I/O channel is tied to the interrupt level of the same number.

Table 7-1. Dedicated Interrupt Locations

Memory Location (Octal)	Interrupt Level
0	0
2	1
4	2
6	3
10	4
12	5
14	6
16	7

INTERRUPT PROCESSING

Certain instructions are provided for the express purpose of processing interrupts, accepting the data input or sending output as required, etc. The BRM-BRD pair is used for bracketing subroutines that handle all varieties of interrupt processing requirements; the IO and INR instructions provide for processing interrupts without resort to a subroutine (single instruction processing); and IOID allows the unit address as well as the channel address to be used for discriminating between routines to process an interrupt from a given device. With certain qualifications as described below, these instructions are associated with the three kinds of interrupt servicing that are available: 1) normal interrupt processing, 2) single instruction processing, and 3) indirect interrupt processing.

Normal Interrupt Processing

A simple routine is used for servicing many kinds of devices. First, a BRM instruction (located at the interrupt location as shown in the table) is executed, to save the contents of the program counter and the condition codes, then transfer to the interrupt servicing routine. The proper service routine is always entered since each interrupt is associated with a unique memory location and therefore with a BRM to its own servicing routine. To exit from the routine, a BRD instruction restores the condition codes and program counter and sends a debreak signal to clear the interrupt level. This returns control to the main program at the appropriate point; the main program is therefore unaware of the interrupt processing, except for values in memory and registers which were intentionally altered by the interrupt program.

Single Instruction Processing

Normally when an interrupt occurs the programmer is responsible for clearing the interrupt level (by using a BRD instruction, which issues a debreak signal) at the completion of his service routine. However, if an INR or IO instruction is executed from the associated memory location in response to an interrupt, the interrupt is cleared automatically after the instruction is executed. Any of the eight priority levels may be used in this manner. The limitations of this method are that the INR instruction operates only as a counter of interrupts and cannot identify the source if there is more than one device on that level; similarly, using the IO instruction in this manner precludes daisy-chaining interrupts and I/O devices on that level. Further extension of the interrupt system is facilitated by:

Indirect Interrupt Processing

There are occasions when it is desirable to use more than eight interrupt levels. Using the indirect interrupt capability (IOID instruction), up to 64 sources on each of the eight priority levels can be automatically identified by the hardware and easily serviced by software. This is accomplished by putting more than one device on some level (or levels), assigning a device address to each device, and using an IOID instruction in the corresponding memory location. When an interrupt occurs, the address field of the IOID instruction is altered by a hardware feature that replaces the least significant six bits of the address field with the six-bit device address supplied by the interrupting source. This is the same device address explained under "Device Address" in Section 6. The address thus generated is used to fetch an instruction which is executed. If the instruction thus executed is an IO or INR instruction, it will function just like a single instruction interrupt (see above); otherwise, it will be a BRM to a processing routine as described under "Normal Interrupt Processing" above.

Using the indirect interrupt allows up to 512 devices (8 levels times 64 device addresses) to be uniquely identified by the hardware. Priorities within the level are established by an enable line connected to all the devices on a level: when a device of a given priority generates an interrupt, all devices of lower priority on the level are locked out. If a device of higher priority interrupts after a lower priority device but before the IOID instruction, the higher device preempts the lower device.

Note that the indirect interrupt feature is a hard-wired option for each controller card. If a controller is designed to be used with the IOID instruction, it must always be used with IOID; if not, IOID cannot be used.

INTERRUPT CONTROLS

An interrupt level may be in any of four states: inactive, waiting, active (requesting service), or busy (being serviced). In the inactive state, no interrupt signal has been received

into the level and no signal is currently being processed. In the waiting state, an interrupt has been recorded as entering the system but is not yet in the priority chain for processing. In the active state, the level has been recognized for processing by the hardware and will interrupt the current program as soon as an interruptable point in the program occurs if it is the highest active level. In the busy state, the level has interrupted the program that was being processed and the processor is now performing processing associated with the level.

An interrupt level may be reset, armed, or disarmed. These controls are implemented by the PIR, PIA, and PID instructions, respectively. Use of these instructions controls the states of any or all of the interrupt levels as specified by the programmer. The arm and disarm instructions control the advancement of an interrupt within the specified level from waiting to active. If the level is disarmed, an interrupt request will be recorded as waiting but will not be processed further until the level is armed. If the level is armed when the interrupt occurs, the request will be serviced as soon as an interruptable point in the microprogram is reached (assuming that no higher level interrupt that is armed has occurred before the interruptable point).

The reset instruction (PIR) controls the clearing of the wait, active, and busy states so that if any request has been recorded, it will be cleared and the level will be placed in an inactive state. However, if the level was in the busy state, the associated processing will continue to completion unless interrupted by *any* of the levels.

Note that SYSTEM RESET resets and disarms all interrupt levels and clears out the interrupts on all devices.

NON-INTERRUPTABLE INSTRUCTIONS

For the convenience of the programmer, six instructions prohibit an interrupt from occurring until after the next instruction has been executed. The instructions and reasons are:

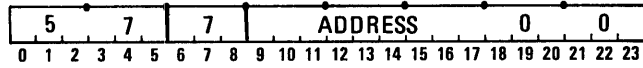
- *XEC, IOID*. These instructions operate by manipulating the TIR (Temporary Instruction Register) directly. In an interrupt were allowed, it would fetch another instruction and load it into the TIR over the XEC or IOID forced instruction. Note that these two instructions differ from branch, skip, and other related instructions, which manipulate the program counter as opposed to the TIR.
- *BRM, BAL*. These instructions branch to subroutines, which sometimes need to disarm certain interrupt levels (e.g., to prevent undesirable reentrancy).
- *PIA*. In implementing interrupt routines that cannot operate reentrantly, it is desirable to issue the BRR or BRD immediately after the PIA is executed.
- *PIR (7002 Processing Unit only)*. This instruction arms all the levels it resets. The programmer may wish to disarm these levels after resetting them.

INTERRUPT INSTRUCTIONS

Interrupt instructions do not affect the condition codes; execution equations are not applicable.

Label IOID Expression

Indirect Interrupt



Description:

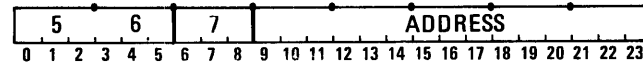
IOID is only valid when executed from a dedicated memory location as a result of an interrupt occurring on the corresponding priority level. At execution of IOID, the 6 least significant bits of the EA are replaced by the unit address of the interrupting device. The newly constructed address is then used to fetch an instruction that is executed.

The instruction must be an INR, IO, or BRM for proper execution. If INR or IO is used, the interrupt level is automatically cleared after execution. If the instruction is BRM, the user must clear the level by leaving the service routine with a BRD. IOID is a non-interruptable instruction.

The assembler treats IOID like any other memory reference instruction and will not assign it to the proper boundaries. The programmer must use absolute addressing with this instruction. See Section 8.

Label PIR Expression

Priority Interrupt Reset



Description:

PIR clears out the wait, active, and busy states of the priority interrupt system according to bits 16-23 of the contents of the effective address. Levels with a 1 in the corresponding bit are reset, and levels with a 0 are not affected.

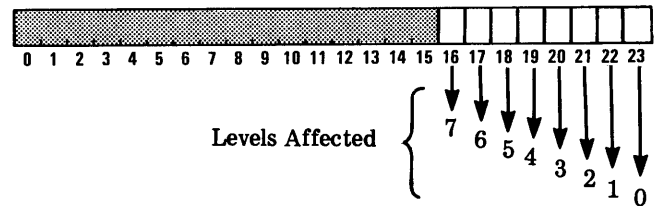
On the 7001 Processing Unit, this instruction is intended for diagnostic purposes only since it does not clear the interrupt request from the peripheral controller. This version of the instruction allows an interrupt immediately after its execution.

On the 7002, PIR resets the interrupt system and the controller. It operates by giving a signal that allows any interrupts in wait (on any selected level) to go active, then acknowledges the interrupt, which clears it and allows another to be generated. The instruction loops in this manner until no more interrupts are generated. At this point the instruction resets the wait, active, and busy states, then exits. Note that PIR arms all the levels that it resets, even if they were disarmed before execution; the user may

wish to disarm these levels after resetting them. It will also clear out any levels that were armed at time of execution; the user must disarm all the levels in which the wait state is not to be reset. The active state of all levels will always be cleared. This new version of the instruction is non-interruptable; an interrupt will not be loaded into the priority chain until completion of the following instruction.

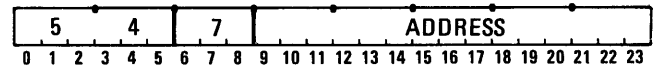
This instruction is used primarily during startup. In general, it is recommended that after a PIR, the following procedure be followed in starting up an input device. First (immediately after the PIR), disarm the level of the device. Next, take status on the device and if an input data ready condition is indicated, issue an IO input (the data can be ignored). Next, the level can be armed and input started.

The contents of the effective address are interpreted as follows:



Label PIA Expression

Priority Interrupt Arm

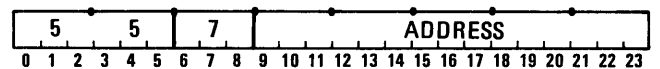


Description:

PIA arms the selected levels of the priority interrupt system according to bits 16-23 of the contents of the effective address. All levels with a 1 in the corresponding bit are armed, and levels with a 0 are not affected. Format for the [EA]₁₆₋₂₃ is described under PIR. If an interrupt request occurs during the execution of this instruction, it is not loaded into the priority chain until the completion of the following instruction.

Label PID Expression

Priority Interrupt Disarm



Description:

PID disarms the selected levels of the priority interrupt system according to bits 16-23 of the contents of the effective address. All levels with a 1 in the corresponding bit are disarmed, and levels with a 0 are not affected. Format for [EA]₁₆₋₂₃ is as described under PIR.

Section 8

Assembly Language Programming

GENERAL

The System IV/70 CODE assembly language is a general purpose programming language, providing the programmer full use of the computer's capabilities without need for binary or octal coding. Prominent features include:

- All input in alphanumeric form, including instructions and data.
- Symbolic addressing.
- Separate assembly of easily-managable program routines with inter-routine linkage automatically handled using non-local symbols (virtuals).
- Absolute and relocatable code with provisions for intermixing.
- A full set of assembler directives for data and assembler control.

ASSEMBLER PROGRAMMING

The assembly language programmer writes his program using symbolic code consisting of alphanumeric mnemonics, symbols, and data. This symbolic code (source program) is converted into binary code by the assembler and loader programs, and executed.

Programs and Routines

Programs are developed by breaking them up into routines which can be composed and checked independently. These routines are defined as blocks of code that may be assembled: the only restriction on a routine is that it must have an END statement as its last statement. (Within a routine, of course, the programmer may have any number of sub-routines.) The assembler program processes a source routine and outputs relocatable code and a listing.

Relocatable code is binary-symbolic code written on an external medium, such as a disc, cards, or paper tape, that may be loaded into the computer for execution. The *listing* is a printed list of the relocatable code in octal form and the source routine.

One or more routines may be combined to make up a program, which is also terminated by an END statement, but which also must meet the criterion of being executable: this means that all inter-routine cross references, or linkages, must be resolved. These linkages are handled using non-local symbols (virtuals) and ENTRY statements; this concept will be explained in detail later. As the routines of his program are assembled, the programmer correlates them using the routine listings, which include printouts of the virtuals and ENTRY statements.

Program Elements

The fundamental program element is the *statement*, which may be constructed with the aid of *symbols* and *expressions*.

STATEMENTS

Statements are of three types: machine instructions, assembler instructions (also called directives), and comments.

Machine instructions each generate a single line of machine code (binary code) that will be executed by the computer. They are explained in detail in Sections 3 through 7.

Assembler directives are instructions to the assembler program to perform various tasks such as data definition, conditional assembly, etc. They are explained under "Assembler Instructions" in this section.

Comment statements are programmer documentation aids that appear only in the assembler output listing. They are designated by an asterisk (*) in the first character position of the statement.

Both machine instructions and assembler directives contain a mnemonic operation code and an operand or operands. They may also contain a symbolic label and comments.

SYMBOLS

General

Many machine instructions and directives refer to addresses in memory, usually for the purpose of fetching or storing data. Rather than being forced to keep track of these addresses, the programmer may refer to each of them with a symbol which is meaningful to him. These symbols are symbolic addresses; other symbols, such as reserved symbols and symbols defined by the EQU directive are symbolic values. A symbol can be any string of one to six characters, the first of which must not be numeric.

An address symbol is defined when it occurs in the label field (see "Assembler Language Coding") of a machine instruction or a directive; whenever the assembler program encounters a symbol used in this manner it assigns the symbol to the address of the location in memory where the instruction or directive resides. Thus the address in question can be referred to and used by other statements in the routine; whenever the symbol appears in the operand field (see "Assembler Language Coding") of another instruction, the assembler program will supply the address in place of the symbol. It follows that a given symbol may be defined only once in any routine, although it may be used any number of times in operand fields.

In addition to relieving the programmer from the task of keeping track of numerical storage addresses, symbolic addresses also allow routines to be stored and executed in any part of memory without affecting the coding of routines. Although the execution addresses of instructions in such routines are not known at the time of coding, their position relative to the start of the routine is known. These instructions are therefore relocatable since the execution (absolute) address may be produced by adding any desired base address to the relative addresses in the routine when it is loaded. All symbolic addresses are relocatable unless they are defined in a section of a routine designated as absolute by an ORG directive.

Reserved Symbols

Reserved symbols, which may appear only in the operand field, are symbols that identify the eight working registers of the CPU or the \$, which stands for the current value of the location counter in the assembler. The nine reserved symbols and their values are:

Symbol	Value	Symbol	Value
R0	0	X1	5
R1	1	X2	6
RP	2	X3	7
RA	3	\$	Current value of location counter
RB	4		

Non-Local Symbols

Symbols defined and used within a routine are called locally defined; but a symbol may be used within a routine without being defined in that routine. A symbol used in this manner is called a non-local symbol or a virtual; these symbols are used to perform linkage between routines of a program. This linkage function facilitates the combining of easily written routines into complicated programs.

A program may contain any number of these virtual symbols, but each of them must be resolved for program execution to take place. A virtual is resolved by the loader program through the use of the symbol in the operand field of an ENTRY directive in the source routine where the symbol is defined. The assembler outputs a notice of all the virtuals and ENTRY statements within a routine to notify both the programmer (via the listing) and the loader (via the binary-symbolic code) of the status of all non-local symbolic information in the routine.

Examples: VIRT is a symbol defined in some other routine.

BRA	VIRT	Example 1
BRM	VIRT	Example 2
STA	VIRT	Example 3

Example 1: This routine transfers control unconditionally to another routine. No thought is given at this time to returning control to the first routine.

Example 2: This routine transfers control to another routine, and the second routine can easily transfer control back to the first using a BRR VIRT. This method is particularly useful for subroutine linkage.

Example 3: This routine stores data for later use by another routine.

EXPRESSIONS

The assembler program will evaluate expressions representing addresses and other quantities as required by the programmer. The expression may be a single symbol or quantity, or symbols (except virtuals) and quantities may be combined in an operand field to produce any desired quantity. The operators allowed are + or & (plus), - (minus), * (multiply), and / (divide). Whenever the assembler program encounters such an expression, the program attempts to evaluate it. If the program cannot evaluate the expression, an E error is generated and the program continues. If a non-integer result is obtained, it will be truncated. Any numeric expression with a leading zero will be interpreted as an octal number; other numbers will be treated as decimal. Note that no grouping of terms in an expression is allowed: the program merely evaluates the expression from left to right by combining single symbols or quantities on each side of the operators. The method of evaluation is very flexible: the only restriction being that the expression must be reducible to an absolute quantity plus a term consisting of the starting address of the routine times a coefficient with a value of -1, 0, or +1.

Symbols used in the operand field of specified assembler directives must be defined (cannot be virtuals) before the assembler evaluates the operand. These directives are BES, BSS, EQU, FORCE, IFGT, IFLT, IFNZ, IFZO, ORG, and SKIP. The reason for this restriction is that these directives can affect the value of the location counter in the assembler and they must be resolved before the location counter is assigned values.

The assembler evaluates expressions working from left to right, then stores the results from right to left. Thus, if the quantity is *less* than the largest number that can be stored (37777777 positive or 40000000 negative), it will be stored correctly, right-justified. But if it is *greater* than these constants, it will be truncated on the left and significance will be lost. See DCN for examples.

Assembler Language Coding

The coding of statements in the assembly language can best be understood with reference to the coding form shown in Figure 8-1. Instruction statements may include four fields: *label*, *operation*, *operand*, and *comments*. Each instruction statement must have an operation field and may have label or comment information. The contents of the operand field for a given instruction depend on the nature of that instruction. Both machine and assembler instruction statements may have from one to four subfields in their operand fields.

For machine instructions, this information is detailed in Sections 3 through 7 and summarized in Appendix C. For assembler instructions, see "Assembler Instructions" in this section and Appendix B. Note that a line of code with an asterisk (*) in column 1 is treated as comments, is not an instruction, and appears only in the assembler output listing; also any characters occurring after the first blank following column 14 are treated as a comment.

LABEL FIELD, COLUMNS 1-6

A label is a symbol consisting of one to six characters of which the first must not be numeric. It represents symbolically the machine location of an instruction or an item of data; this location is called a symbolic location. A symbol may be written anywhere in the label field.

OPERATION FIELD, COLUMNS 8-13

Each machine or assembler instruction has one or more mnemonics assigned to it; such a mnemonic may be written anywhere in the operation field. An asterisk written at the end of the mnemonic for address modifiable instructions indicates indirect addressing. See Sections 3 through 7 for machine instructions and "Assembler Instructions" below for assembler directives; see Appendices B and C for a summary.

OPERAND FIELD, COLUMNS 15 TO FIRST BLANK

The operand field contains an operand or operands that are peculiar to each instruction. The operand information must be written starting in column 15 and must not contain a blank except for character string definition in a DCA directive. With this single exception, the first blank after column 14 marks the end of the operand field. The operand for memory reference instructions consists of an expression representing a memory address followed by an index register symbol when indexed. Non-memory reference instructions may contain source register, destination register, byte control, or count control operand subfields depending on the instruction. These operands must be written in the order given and must be separated by commas. See the descriptions of the instructions and Appendices B and C for details.

Examples:

LABEL RCR	R1,RA,7,8	SEE NOTE 1
STA	LOC1	SEE NOTE 2
STA*	LOC2,X1	SEE NOTE 3
LOC1 PZE	012345	SEE NOTE 4

Note 1: This instruction will be executed as a register copy and rotate of R1 into RA, with a shift count of 8 and a byte control of 7 (all bytes). The instruction will be placed in an appropriate location in memory, and the label, LABEL, takes on the value of the address location. "SEE NOTE 1" is a comment and would appear in the listing only.

Note 2: This instruction will store the contents of RA into LOC1, a symbolic memory location. Note that the label or

operation information may appear anywhere in their allocated fields; only the operand information *must* begin in column 15. This instruction will appear in the next memory location following the RCR instruction. Since no label is supplied, this location will not appear in the symbol table, but it could be referenced using LABEL+1 as a symbolic location.

Note 3: This instruction is similar to the STA above, except that indexing (the tag field, "X1") and indirect addressing (*) are specified.

Note 4: A typical assembler instruction. The location LOC1 is assigned and its first nine bits zeroed, and octal 12345 is entered into its 15 least significant bits. Assembler instructions are coded in the same manner as machine instructions.

Absolute and Relocatable Code

In the context of assembly programming, the programmer may think of data and instructions as being absolute (i.e., he expects them to be loaded in the locations specified), or relocatable (i.e., he expects them to be loaded anywhere in memory as a block).

The use of relocatable code covers the majority of programming cases; the principal exception is the absolute code demanded by dedicated areas within memory (see Table 3-4 "Dedicated Memory Locations"). For example, the programs that generate and store characters to be displayed in the video display areas may all be relocatable, but the symbols pointing to the displayable character storage locations must be absolute. Relocatability is determined by use of the ORG directive.

If no ORG statement precedes a block of code, the entire block is relocatable, and the symbolic addresses defined in the block are relocatable. A block of code is also relocatable if it is preceded by an ORG statement containing a relocatable operand expression. Relocatable blocks of code will be assembled relative to an assumed 0 starting location and loaded with a relocation bias calculated by the loader for the most efficient use of memory. Note that the relocation bias is added not only to the relative locations of instructions and data, but also to the relocatable address fields within instructions.

If an ORG statement is used with an absolute expression, the following block of code will be assembled and loaded relative to the absolute expression as a starting location. However, the relocation bias will still be added to relocatable address fields within instructions.

The ORG statement is designed to promote easy intermixing of absolute and relocatable code. Whenever a symbol is encountered in the label field of an ORG statement, the symbol is set equal to \$ (the current value of the location counter), then \$ is set equal to the value of the operand expression. Later, when an ORG statement appears with

the same symbol in its operand field, \$ is set equal to its earlier value, and the program proceeds. Only a block of code between an ORG statement that designates absolute code and one that designates relocatable code is treated as absolute; the rest of the routine is relocatable.

For further information on the operation of the Assembler and Relocatable Loader, see "Disc Operating System (DOS) Reference Manual" document SIV/70-50-1.

SEQUENCE OF EVENTS

The sequence of events in generation and execution of an assembly language program is as follows: first the programmer writes his program (i.e., one or more routines) in alphanumeric form with mnemonic operations, symbolic addresses, etc. Second, the assembler program makes two separate passes upon each routine.

During the first pass, symbols used as labels in the program are tabulated in a symbol table along with the corresponding addresses that define the symbols. Assembler directives pertaining to assignment of memory locations are also completely processed so that storage can be allocated for instructions and data.

The second assembler pass generates and outputs binary-symbolic object code representing data, machine instructions, virtuals and their locations, ENTRY statements identifying symbols used outside this routine, and sufficient additional information to enable the loader program to locate the instructions and data properly for execution. The second pass also outputs the information needed to generate an assembly listing, which forms the documentation of the routine as generated by the computer.

An assembly listing is illustrated in Figure 8-2. The first column on the listing flags any errors (e.g., the E for an evaluation error at location 04006). The next column shows the octal memory location, if any, assigned to each statement by the assembler. Note that the ENTRY, ORG, and comment (*) statements receive no address assignment. The next column shows the contents of the corresponding memory location, in octal. The exception to this is the ORG statement, which shows 00000000. The rest of the columns show the symbolic program as entered by the programmer. This is the same information contained on the coding form, Figure 8-1. Label, operation, operand, and comment information are listed in that order. At the bottom of the listing, virtual symbols are listed along with associated address information.

The loader program, used with the assembler, takes the binary-symbolic relocatable code generated by the assembler and allocates instructions and data to locations where they can be executed. Unless directed otherwise by the programmer or assembler program, the loader always loads each routine into the lowest- or highest-numbered memory locations available, to provide the most efficient possible use of memory. Before program execution is started, the loader will resolve all virtual symbols, assigning the appropriate linkages between routines.

ASSEMBLER INSTRUCTIONS

General

The assembler instructions (directives) perform program functions that are not a part of the machine instruction set. The directives perform data definition and storage allocation functions, plus various assembler control functions. They are discussed as *Data Control* and *Assembler Control* directives.

00000	00004010		ENTRY	Z	
04000	00000000		ORG	04000	
04000	06703700	START	RCPY	R0, RA	CALCULATE
04001	05000002		LD1	2	SUM OF YS
04002	13204014	LOOP	ADA	Y+3, X1	
04003	75004002		BC1	LOOP	
04004	43004010		STA	Z	ENTRY USED
04005	72004005		BRA	MAIN	RETURN TO MAIN
E04006	72000000		BRA		E ERROR
04007	00004007		HLT	\$	CONVENTIONAL HALT
04007	00000000	*	COMMENT	CARD -	PROGRAM CONSTANTS FOLLOW.
04010	00000000	Z	PZE	0	ENTRY DEFINED
04011	00000005	Y	DCN	5	Y1
04012	77777776		DCN	-2	Y2
04013	00000003		DCN	3	Y3
04014	77777775	I	DCN	-3	INDEX CONSTANT
MAIN	VIRTUAL 04005				
04014	72004000		END	START	NEED END TO ASSEMBLE

Figure 8-2. Assembler Output Listing

Data Control Directives

These directives define symbols as the storage location for numbers, character strings, and values of expressions; allocate storage space; and equate symbols to values of expressions. Any time the assembler program encounters an expression in the operand field of one of these directives, the program attempts to evaluate it. Quantities thus derived will be stored in binary form, truncated to fit into a computer word. There are *whole word definition*, *part word definition*, *storage allocation*, and *symbol definition* data control directives.

WHOLE WORD DEFINITION

Whenever a label is used with one of these directives, the label is set equal to the address where the constant is stored. This contrasts with the EQU directive, which sets the label equal to the value of the expression.

Label DCA 'c₁c₂c₃...c_n'

Define Constant ASCII.

The characters between the delimiters (shown here as ') are treated as a character string, converted to ASCII, and packed three-to-a-word in memory. The delimiter can be any ASCII character but must not appear in the string. Unfilled character positions on the right end of the last word are filled with blanks unless a zero follows the second delimiter. In this case, the last word will be filled with zeros. The label field is optional; if a label is given, it will be assigned to the location of the first three characters.

Label DCN Expression

Define constant numerical, integer.

Evaluates the expression and stores it as a 24-bit quantity (or 23 bits + sign). The precision of the quantity stored is equal to the capacity of a single memory word; i.e., 23 bits. Thus for a decimal number, the maximum precision is just under seven digits. Also, the user may enter a number in octal (an octal quantity is indicated by a leading zero).

Note that if a label is used with a DCN, the label is assigned to the *location*, not to the *quantity*. Examples:

Assembler Input	Assembler Output
DCN 04000000	40000000
DCN -2	77777776
DCN 012*256+013*256+014	02405414
DCN 1	00000001
DCN 01122334455	22334455

Label DCN ±n₁n₂n₃...n_pB±m₁m₂...m_q

Define constant numerical, fixed point fraction.

Converts the decimal integer and fraction represented to the corresponding binary number and places the result in a 24-bit word in memory. The scaling of the result is determined by the absolute magnitude of the decimal input and

by the value of M, the number following the B. In general, M should be greater than the number of bits required to express the integer part of the decimal input. M specifies the number of bit positions between the binary point and the normalized binary point location; i.e., between bit positions 0 and 1 (M is + for right and - for left displacement of the binary point). If the input decimal number is negative, the binary result will be calculated as if a positive number had occurred, then the result will be 2's complemented. A capital B must appear as shown, if the binary point is used. Examples:

	Assembler Input	Assembler Output
PI	DCN 3.1416B2	31103774
	DCN 14.375B4	34577777
	DCN -14.375B4	43200001
LOGE	DCN .4343B-1	33627110

Label DCS ±.n₁n₂n₃...n_pE±m₁m₂...m_q

Define constant single-precision floating point.

Enables the programmer to define constants for use by the floating-point arithmetic instructions. The decimal number in the operand field is converted to a standard two-word floating-point number (see "Floating Point Data" under "Formats" in Section 3). The exponent in the operand field must be signified by an E.

Label DCD ±.n₁n₂n₃...n_pD±m₁m₂...m_q

Define constant double-precision floating point.

Enables the programmer to define constants for use in extended-precision floating-point subroutines. The decimal number in the operand field is converted to an extended-precision three-word floating-point number (see "Floating Point Data" under "Formats" in Section 3). The exponent in the operand field must be signified by a D. The label is assigned to the first word. Example:

Assembler Input	Assembler Output
FORCE 1	
DCD +.142857142857D0	00141 11111040
	00142 22222222
	00143 77777776

PART WORD DEFINITION

These directives load a constant into the first nine bits of a memory location and operand information in the remaining bits. They are used in constructing instructions, reserving locations, generating constants, etc.

Label PZE Expression

Prefix plus zero.

Puts zeros in the first six bits of the word and the value of the expression in the last 15 bits.

The PZE directive enables the programmer to assign a label to a memory location, to mark the beginning of a BRM subroutine, and to construct the *address part* of a memory reference instruction. For this use, an asterisk (for indirect addressing) and/or a tag field (for indexing) may be specified; see "Address Modification" in Section 3 for details. If PZE is executed in the object program, a HLT will occur.

Label RPZE S,D,B,C

Register prefix zero.

Puts a prefix of 007_8 in the first nine bits of the word and the value corresponding to the optional source, destination, byte control, and count information in the last 15 bits.

The RPZE directive enables the programmer to construct the *operand part* of non-memory reference instructions. If executed, an LCL instruction will be performed.

Label MZE Expression

Prefix minus zero.

Puts a prefix of 777_8 in the first nine bits of the word and the value of the expression in the last 15 bits.

The MZE directive enables the programmer to generate a negative data word; e.g., generation of tables for the character move instructions. If executed in the object program, a HLT will occur.

STORAGE ALLOCATION

These directives allocate blocks of storage locations.

Label BSS Expression

Block starting with symbol.

Reserves a block of memory locations whose length equals the value of the expression, and assigns the label to the first location in the block. The value of \$, the location counter, is increased by a number equal to the length of the block.

Example:

	ORG	0100	Sets location counter to 100_8 .
AA	BSS	020	Defines AA as location 100_8 incrementing the location counter by 20_8 words, thus changing its value to 120_8 .
	LDB	VALUE	This instruction is placed in the location immediately after the 20_8 reserved words, namely in location 120_8 .

Label BES Expression

Block ending with symbol.

Reserves a block of memory locations whose length equals the value of the expression; in doing so it increases the value of \$ by that amount. If a label is supplied it is assigned to the word beyond the last location in the storage area and identifies the block of storage.

SYMBOL DEFINITION**Label EQU Expression**

Symbol equals expression.

The label is given the value of the expression, with an upper limit of 15 bits. The expression must not be a virtual. Note that if an EQU is used with a quantity greater than 15 bits, the most significant bits will be lost (i.e., the expression is treated modulo 32,768). However, if a quantity defined in an EQU statement is later used in an expression, the nine most significant bits will fill with arithmetically non-significant bits; i.e., 0's for a positive number and 1's for a negative number. Note that this instruction does *not* assign a symbol to a memory location. The label field must always be used.

Example:

Assembler Input			Assembler Output
S1	EQU	-12	77764
S2	DCN	S1	7777764

Assembler Control Directives

These directives control the assembly process by indicating starting and ending points, controlling the location counter, and making the assembly of groups of statements conditional upon programmer-chosen conditions. There are *counter control*, *conditional assembly*, *linkage control*, and *miscellaneous* assembler control directives.

COUNTER CONTROL

The counter control directive selects absolute or relocatable starting locations.

Label ORG Expression

Origin setting of location counter.

Sets the label equal to the current location counter value (\$), then sets \$ equal to the value of the expression. The expression must not be a virtual. If no label is specified, ORG merely sets \$ equal to the value of the expression.

If the expression is absolute, the following code will be assigned to absolute locations; if the expression is relative, the following code will be assigned to locations relative to the start of the routine. (See "Absolute and Relocatable Code" for details.) A symbolic address is absolute if the symbol is defined by code assigned to absolute locations, otherwise the symbol is relative.

Example:

```
BB    ORG    06    Assigns the label BB to the
                    current contents of the loca-
                    tion counter and then sets the
                    location counter to 68.

        LD2    INDEX This instruction is stored in
                    location 68.
```

The same effect would have been produced by:

```
BB    BSS    0
        ORG    06
        LD2    INDEX
```

CONDITIONAL ASSEMBLY

Conditional assembly directives make the assembly of part of a routine depend on programmer-determined data items or relations. These directives are widely used for the implementation of control cards. They will not skip beyond an END instruction. No error message is printed if one attempts to do so.

Label SKIP Expression

Skip assembly on greater than zero.

Inhibits the assembly of the next N statements following the SKIP instruction, where N is the value of the expression. There is no skipping if $N \leq 0$.

Example: Changing the TRUE statement to EQU 0 causes a program change.

```
TRUE EQU 1
SIM EQU TRUE
        LD1 TRA
        ---
        ---
TRA BSS 0
        SKIP 1-SIM
        DCN -01000 Count 01000
        DCN -02000 Count 02000
```

Label IFGT Expression, LABEL

Skip assembly if greater than zero.

If the value of the expression is greater than zero, the assembly of statements is suspended until LABEL is encountered in columns 1-6 of some later statement. LABEL must be supplied. Note that a label in the label field of a conditional assembly statement is legal only if it is used as an operand-field LABEL in another conditional assembly statement. The LABEL can be from one to six characters, and does

not have to be a legal symbol. This allows skipping to a comment statement as shown in the following example:

```
A    DCN    $+3
B    DCN    -15
        IFGT A+B,*NEXT
        ---
        ---
*NEXT          Will be assembled if it's an instruction
        ---
```

Label IFLT Expression, LABEL

Skip assembly if less than zero.

Operates exactly the same as IFGT, except that assembly is skipped if the value of the expression is negative.

Label IFZO Expression, LABEL

Skip assembly if zero.

Operates exactly the same as IFGT, except the assembly is skipped if the value of the expression is zero.

Label IFNZ Expression, LABEL

Skip assembly if not zero.

Operates exactly the same as IFGT, except that assembly is skipped if the value of the expression is not zero.

LINKAGE CONTROL

The linkage control directive allows linking of routines to form a complete program.

ENTRY Symbol

Enter symbol value in other routines.

Notifies the assembler that the symbol in the operand field is intended to be used as a virtual link by other routines. The symbol must be defined by its appearance in the label field elsewhere in the same routine. The label field must be left blank.

Label EOP

End of program; link to library.

Terminates the assembly of a program or the processing of a relocatable library. If the label field of the EOP directive contains a symbolic name, the relocatable loader will use this name as a library name in resolving virtuals. Thus, if there are unresolved symbolic references in the user program at load time, and if the name in the label field of the EOP is the name of a relocatable file on the disc, the loader will search the relocatable file and fetch any routines that are needed to resolve the virtuals. This mechanism can also be used for linking relocatable libraries together. See "Disc Operating System (DOS) Reference Manual", document SIV/70-50-1 for details.

MISCELLANEOUS

END Expression

End of routine or program.

Terminates the assembly of a routine or program. An expression may be included in the operand field of the END statement. Then, depending on the parameters given at load time, the value of this expression or the lowest address which is used by the program and contains data will be taken as the starting location of the program. With the current loader an attempt to load a program at 0 will fail; if a program is absolutely originated it must be placed at 1 or higher.

Note that an END statement must be given before a routine can be assembled — without the END statement the assembler will not process the routine. No label field may be used with an END statement.

FORCE 0 or 1

Force an even or odd starting location.

Forces the next location to even or odd, depending on whether the contents of the operand field are even (0) or odd (1). 1 is added to the least significant bit if needed. The label field must be blank. The FORCE directive is commonly used to force even boundaries for I/O select words, load and store double instruction data, etc.

ERROR CONDITIONS

The assembler detects four kinds of errors: erroneous operation code (O), doubly defined symbol (D), evaluation error (E), and symbol table overflow. The relocatable loader detects various errors as outlined in the "DOS Reference Manual" document SIV/70-50-1.

ERRONEOUS OPERATION CODE (O)

If the assembler detects an entry in an operation field (columns 8-14) that is not in its op code table, an O will be printed and 00000000₈ assembled. Assembly will continue, but a halt would be executed.

DOUBLY DEFINED SYMBOL (D)

If the assembler detects the same symbol in two label fields within a routine, a D will be printed. Assembly will continue, but the second entry will not appear in the symbol table.

EVALUATION ERROR (E)

If the assembler encounters an expression it cannot evaluate, an E will be printed and 00000₈ assembled. Assembly will continue, but a halt would be executed.

SYMBOL TABLE OVERFLOW

If too many symbolic labels (current limit = 360₁₀) are encountered and the symbol table fills, assembly will halt. This is the only error condition that will terminate assembly.

Section 9

System Operation

INTRODUCTION

This section discusses operation of the computer with emphasis on use of the control panel. The control panel has various system control functions; its controls and indicators are illustrated in Figure 9-1 and described in Table 9-1. Functions that may be performed using the control panel include:

- Displaying the contents of a register.
- Displaying the contents of a memory location; stepping through a sequence of memory locations.
- Altering the contents of a register; altering the condition codes.
- Altering the contents of a memory location.
- Executing an instruction; stepping through the sequence of operation of a program.
- Automatically executing an instruction repeatedly.
- Initializing the system by bootstrapping a program.
- Halting operation and clearing certain error conditions.
- Automatically entering the contents of the keys into a register.

MANUAL DATA DISPLAY AND ENTRY

Register Data Display

During normal operation in the AUTO mode, the DISPLAY switches usually remain in the TIR position (000). When the AUTO/MANUAL switch is set to MANUAL, the contents of the instruction register (i.e., the next instruction due for execution) are displayed immediately. To display the contents of any working register (RA, RB, RP, X1, X2, or X3), select the register on the DISPLAY switches according to the table on the control panel.

Note that, in MANUAL mode, the status bits are stored in the contents of RP, bits 0-5. Whenever RP is selected, these bits will be displayed. The status bits are stored as follows:

RP Bit	Status Bit
0	Stop (always 1 in MANUAL mode)
1	Malfunction (Main Memory parity error)
2	Overflow condition code
3	Zero condition code
4	Minus condition code
5	Carry condition code

During operation in AUTO mode, the DISPLAY switches can also be left in the RP position (010). If this is done, bit 0 (Stop) will come on brightly if a HLT is executed and bit 1 (MM) will come on brightly if a memory parity error occurs.

Memory Data Display

If MEM (001) is selected on the DISPLAY switches in MANUAL mode, the contents of a memory location will be displayed. The location thus displayed will be the address indicated by the program counter; i.e., bits 9-23 of [RP]. If the contents of the program counter are changed, a different location will be displayed (see "Register Data and Condition Code Entry," below). If the STEP switch is activated with MEM selected, one will be added to the program counter, and the contents of the next sequential memory location will be displayed.

Register Data and Condition Code Entry

A register whose contents are being displayed in MANUAL mode may be altered using the 24 keys across the bottom of the control panel and the LOAD switch. Whenever the LOAD switch is activated, the contents of the keys will be transferred to the contents of the register currently being displayed. The new contents of the register will be displayed immediately.

This feature may be used to alter the contents of the four condition codes. First, RP is displayed, Second, the contents of RP bits 9-23 are entered into the keys so that the program counter's contents will not be lost. Next, the new condition code settings are entered into key positions 2-5. Last, the LOAD switch is activated. By this method, any combination of condition code settings may be entered.

Note that use of RP in MANUAL mode is one way to reset the overflow condition code; this CC is otherwise normally reset using the BOF instruction under program control. For details refer to the BOF, BRR, and Decimal Option (DADD) instructions in Sections 4 and 5.

Memory Data Entry

With the DISPLAY switches on MEM, the keys and the LOAD switch may be used to alter the contents of any memory location. The procedure is the same as for altering the contents of a register with the provision that the contents of the program counter are used to select the memory location to be displayed and changed. If the contents of the

program counter are changed to alter the contents of the corresponding memory location, record the contents of the program counter before changing it to allow the interrupted point in the operating program to be restored.

PROGRAM EXECUTION

Automatic Execution

In the normal mode of computer operation, the AUTO/MANUAL switch is in the AUTO position and the computer is executing instructions under control of the computer program. This mode of operation is changed only by moving the AUTO/MANUAL switch to the MANUAL position.

Note that it is impossible to enter the AUTO mode when the DISPLAY SELECT switches are set to MEM (001). Conventionally, the DISPLAY SELECT switches will be set to TIR (000) before AUTO is selected.

Manual Execution

With the console DISPLAY SELECT on TIR in MANUAL mode, the contents of the instruction register will be displayed; this is the next instruction destined for execution in the program sequence. If the STEP switch is activated, this instruction will be executed and the program counter incremented, just as in AUTO mode. Also, the next sequen-

tial instruction will be fetched and stored in the instruction register. If the STEP switch is activated again, this next instruction will be executed also; this sequence may be repeated indefinitely.

These executions are identical to executions in AUTO mode: tests, branches, register and memory changes, etc., will all occur in the usual manner. The only difference is that a Stop will occur after the execution of each instruction. These changes may be examined in detail from the control panel as outlined in the paragraphs above.

Similarly, the contents of the instruction register may be altered using the keys and the LOAD switch before executing using the STEP. If this is done, record the contents of the instruction register and the program counter so that you may restore the halted program when the console intervention is complete.

Repeated Instruction Execution

In some testing situations, it is desirable to automatically execute an instruction an indefinite number of times. The REPEAT switch is provided for this purpose. To use the REPEAT switch, proceed as follows:

- a. Start in MANUAL mode.

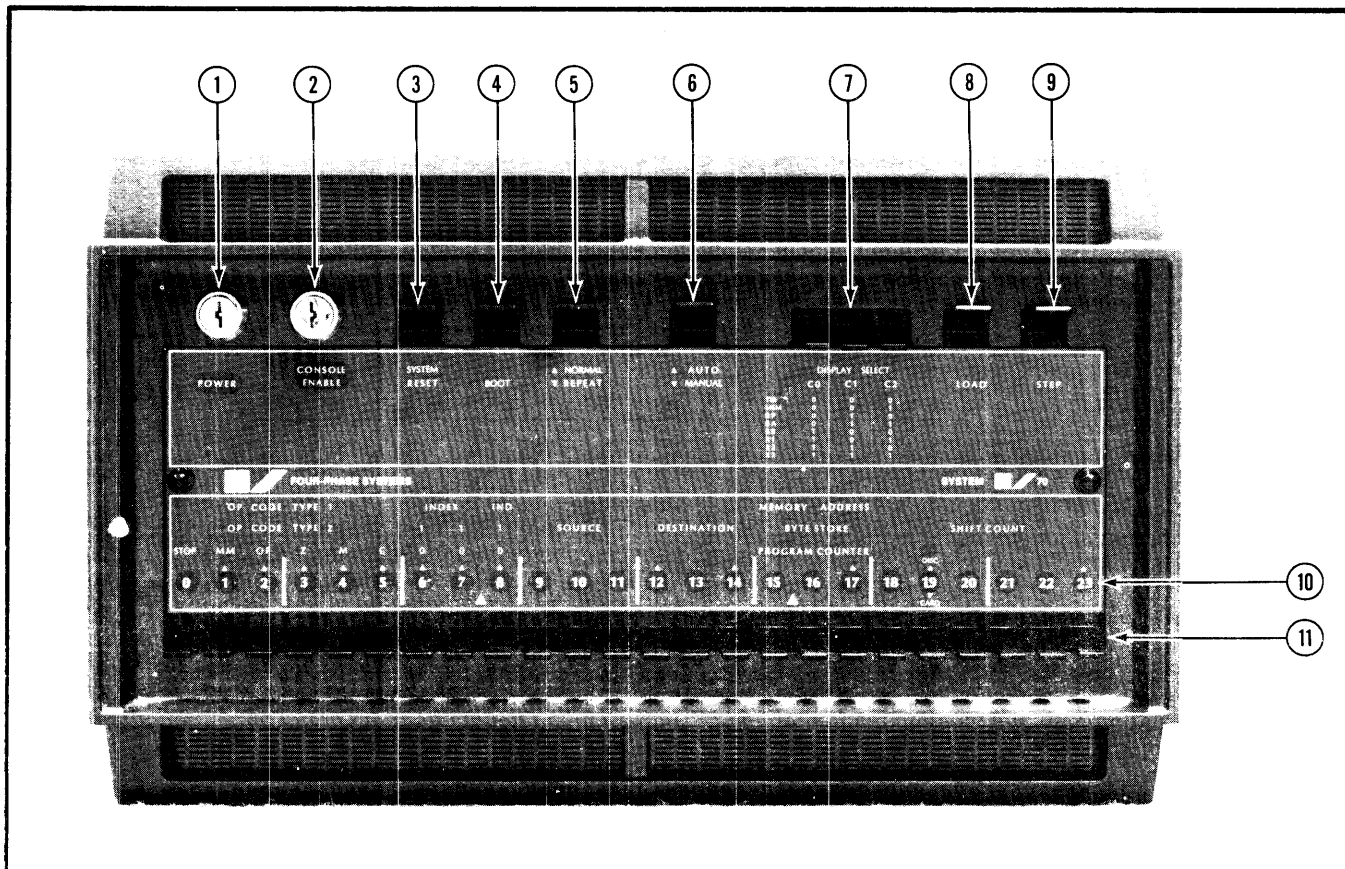


Figure 9-1. Console Controls and Indicators

Table 9-1. Control Panel Controls and Indicators

Figure 9-1 Index No.	Control or Indicator	Function
1	POWER	Applies power to the computer and built-in peripheral devices, such as data sets. Whenever the POWER is turned on, the system reset mode is forced and can not be cleared for 2-3 seconds. Activated when key is turned clockwise.
2	CONSOLE ENABLE	Locks out the functioning of all the console switches, except the keys when key is turned counterclockwise.
3	SYSTEM RESET	Forces the computer into the system reset mode. Resets all interrupts and I/O flip-flops, and forces the computer into a no-operation loop which can only be cleared by activating the STEP switch (9). May be used to clear certain error conditions, such as execution of XEC \$ or an attempt to send a non-existent unit address, but not intended for clearing normal programming errors such as closed program loops; the MANUAL mode is intended for clearing this kind of problem. Activation of SYSTEM RESET also disables the memory parity checking circuits. Momentary switch, active down.
4	BOOT	Enables the operator to bootstrap load the computer with a single switch setting. The use of this switch requires that the DISPLAY SELECT switches all be set down (000 = TIR selected), and that an appropriate constant be set into the 24 keys depending on the peripheral input device. The switches are normally left in these positions during operation. Momentary switch, active down. 7001 Processing Units shipped before May 1972 used an INTERRUPT switch in place of the BOOT switch. It initiates a programmed interrupt from the control panel. Momentary switch, active down.
5	NORMAL/REPEAT	May be used to repeat an instruction; see "Repeated Instruction Execution" in this section. Two-position switch; NORMAL up, REPEAT down.
6	AUTO/MANUAL	Selects the two principal modes of computer operation. In AUTO mode, the computer is executing programs at its normal speed under its own control. In MANUAL mode, the computer is under control of the control panel switches. Various paragraphs in this section explain the uses of the MANUAL mode. Two-position switch; AUTO up, MANUAL down.
7	DISPLAY SELECT	Selects the Temporary Instruction Register (TIR), the working registers (RP, RA, RB, X1, X2, and X3) or a MEMORY location for display or alteration. The uses of these switches are explained in this section. Three two-position switches; one up, zero down.
8	LOAD	Loads the register or memory location selected using the DISPLAY SELECT switches. The uses of the LOAD switch are detailed in this section. Momentary switch, active down.
9	STEP	Used to clear the system reset mode, to step through memory locations, or to execute a program step-by-step; works in MANUAL mode only. Momentary switch, active down.
10	Lights	Displays the contents of B3, the computer's main data bus. In MANUAL mode the contents of the memory location or register selected by DISPLAY SELECT are kept loaded on B3 by the microprogram and thus displayed.
11	Keys	Used to alter the contents of a selected register or memory location under control of the LOAD switch, or the ECS instruction. Two-position switches; one up, zero down.

A066B

- b. Set the keys to the op code, mod field, and operand field of the instruction desired (see Sections 3, 4, 5, 6, and 7 for complete information on any instruction).
- c. Select TIR on the DISPLAY SELECT switches.
- d. Press the LOAD switch.
- e. Set NORMAL/REPEAT switch to REPEAT.
- f. Press the STEP switch.

The instruction will repeat indefinitely. To clear the REPEAT mode, set the REPEAT switch to NORMAL. The computer will now be in MANUAL mode.

This procedure can be used for clearing memory to zero or storing any other constant. Just use the SPR instruction (45700000 into the TIR) after placing zero bits (00000000) or another value intended for repetition into RA.

INITIALIZING THE SYSTEM

Whenever power has been off, the system must be initialized before operation can begin. The normal sequence of events is as follows: turn power on (see "Turning Power On"). This initializes the system reset mode automatically and prevents clearing of the system reset mode for 2-3 seconds so that the I/O system may initialize itself, certain capacitors may charge, etc. Next, the desired program is loaded into the system using the bootstrap loading procedure (see the following procedures).

Turning Power On

The procedure for turning the power on varies depending on system configuration. Proceed as follows:

- a. If the system uses an 8701 Mounting Cabinet but no 7071 Channel Adapter, turn on circuit breaker at the bottom rear of mounting cabinet by placing it in the up position.
- b. If the system is connected to a Channel Adapter and power is controlled by the Channel Adapter, press POWER ON button on the Channel Adapter. The computer POWER indicator should light.

NOTE

Power to systems using a Channel Adapter is controlled by either the 360/370 or the POWER ON and OFF buttons on the Channel Adapter. The key-operated POWER switch on the computer is left in the ON position at all times. If power is controlled by the 360/370, the power will be on when the 360/370 is on.

- c. If no Channel Adapter is attached, turn POWER key switch on (fully clockwise). The POWER indicator should light.

Bootstrap Loading for Systems With a BOOT Switch

- a. Turn computer power on (see "Turning Power On").
- b. Turn CONSOLE ENABLE key switch on (fully clockwise). The CONSOLE ENABLE indicator should light.
- c. Set AUTO/MANUAL switch to AUTO.
- d. Enter the following boot word into the 24 console keys:

Device	Boot Word
Card Reader	37705101
Disc Drive	37705121
Magnetic Tape Drive	37705221

- e. Select TIR by setting all three DISPLAY SELECT switches down.

- f. Prepare the bootstrap device for loading as follows:

- (1) For the card reader: press ON switch, place the binary deck in the card hopper, and then press START switch.

- (2) For the disc drive: set LOAD/RUN switch to LOAD; after LOAD light comes on, open door at front of unit and install disc cartridge (it must slide in completely without forcing or twisting); close door securely; set LOAD/RUN switch to RUN. The LOAD light will go out and the disc will accelerate. After about one minute, the READY light will come on. Do not perform step *g* until the light comes on.

- (3) For the magnetic tape drive: load the tape drive and place it on-line (refer to "Tape Drive Operating Procedures" in this section).

- g. Firmly press BOOT switch. If this does not work, make sure that AUTO is selected.

NOTE

If the bootstrap device is not ready when the BOOT switch is pressed, the system will hang waiting for it to become ready.

Bootstrap Loading for Systems Without a BOOT Switch

- a. Turn computer power on (see "Turning Power On"). If power is on, perform step *b* and then press SYSTEM RESET to initiate the bootstrap mode at the device controller.

- b. Turn CONSOLE ENABLE key switch to on (fully clockwise). The CONSOLE ENABLE indicator should light.

c. Set AUTO/MANUAL switch to MANUAL and press STEP switch. Two or three seconds may be required from the time of supplying power to the computer before the STEP switch will function.

d. Enter the boot word (37705101 for card reader, 37705121 for disc drive, and 37705221 for magnetic tape drive) into X1 and TIR as follows:

- (1) Enter the boot word into the 24 console keys.
- (2) Select X1 (101) on DISPLAY SELECT switches and then press LOAD switch. The indicators above the console keys should light; if not press SYSTEM RESET and STEP, then press LOAD again. If this fails, make sure that MANUAL is selected before STEP is pressed.
- (3) Select TIR (000) on DISPLAY SELECT switches and then press LOAD switch.

e. Prepare the bootstrap device for loading as follows:

- (1) For the card reader: press ON switch, place the binary deck in the card hopper, and then press START switch.
- (2) For the disc drive: set LOAD/RUN switch to LOAD; after LOAD light comes on, open door at front of unit and install disc cartridge (it must slide in completely without forcing or twisting); close door securely; set LOAD/RUN switch to RUN. The LOAD light will go out and the disc will accelerate. After about one minute, the READY light will come on. Do not perform step *f* until the light comes on.
- (3) For the magnetic tape drive: load the tape drive and place it on-line (refer to "Tape Drive Operating Procedures" in this section).

f. Press SYSTEM RESET and then STEP, and set AUTO/MANUAL switch to AUTO.

MISCELLANEOUS PROCEDURES

Using the Interrupt Switch

The INTERRUPT switch, provided on 7001 Processing Units shipped before May 1972, allows the operator to enter a special interrupt routine at any time. This interrupt operates under control of the priority interrupt system of the computer as described in Section 7. In normal use the control panel interrupt is wired to an interrupt level that causes a trap to a special interrupt routine. This routine could be used for displaying special information on all video screens, for halting operations for the day, or for any other function that might be programmed.

Halting and Clearing Errors

Various error conditions can cause the normally programmed course of operations to stop; also, moving the

AUTO/MANUAL switch to MANUAL, or pressing SYSTEM RESET will cause stops. Use of these two switches plus other techniques for correcting error stops are described here. The error conditions, plus the methods for detecting and clearing them are:

EXECUTION OF A HLT INSTRUCTION OR ILLEGAL INSTRUCTION

The illegal instructions are those with op codes 0707 through 0777 and also Decimal Option instructions on a machine without a Decimal Option installed. The method for detecting such a halt is to leave the computer in AUTO and select RP (010) with the DISPLAY SELECT switches. If STOP (bit 0) comes on in the lights at the bottom of the control panel, this sort of halt has occurred; if not, the machine is looping or hung on I/O. If bit 0 and bit 1 (MM) are both on, see "Parity Error" below. To clear a STOP bit halt, one of two procedures is required:

- If this is an expected halt in the program (e.g., waiting for a quantity to be loaded into the keys or a tape to be loaded), perform any required actions, then move AUTO/MANUAL switch to MANUAL and back to AUTO.
- If this is an unexpected error, such as the program executing data, set AUTO/MANUAL switch to MANUAL, select TIR (000) with the DISPLAY SELECT switches, key in a BRA instruction to a convenient program starting point, press LOAD, then set AUTO/MANUAL switch back to AUTO. In case of unexpected errors, it is good practice to record conditions at the time of the halt ([RP], [TIR] etc.). On Four-Phase supported software, a branch to the starting location of the program is conventionally placed in location 1; this practice is encouraged. A branch to location 1 will then restart the program.

PARITY ERROR (MACHINE MALFUNCTION)

If a parity error occurs, the processing of the program will stop. To check for this condition, select RP (010) with the DISPLAY SELECT switches and see if the bit 1 (MM) light at the bottom of the control panel is on. (The light will be on in AUTO or MANUAL mode.) If so, a parity error has occurred. At this point, it is recommended that the machine be left untouched (power *on*) and a Four-Phase Field Service Engineer should be contacted immediately. If it is desired to clear this condition and proceed with the program, set AUTO/MANUAL switch to MANUAL, press STEP, then go back to AUTO.

PROGRAM LOOP

If the program is in a closed loop, STOP (bit 0) will not display when RP is selected in the AUTO mode. To clear a closed loop, set AUTO/MANUAL switch to MANUAL, key a BRA instruction to a convenient point outside the loop into the TIR, then go back to AUTO. Such a closed loop is usually a program bug, and normal procedure is to

record conditions in the loop, step through it, etc., for diagnostic purposes. Execution of a BRA \$ acts exactly like this kind of loop.

I/O HANGUP, EXECUTION OF XEC \$

Certain I/O problems can hang the processor as well as the execution of an XEC to the current RP location. An example of such an I/O bug is the attempt to address a non-existent I/O controller; i.e., the sending of a non-existent channel or unit address. A hardware failure in the controller circuit can also cause this problem. The symptom is that the machine is hung and will not respond to the AUTO/MANUAL, DISPLAY SELECT, LOAD, or STEP keys. The cure is to select MANUAL, then press SYSTEM RESET and STEP. The machine is now in normal MANUAL mode and error recovery procedures (such as those outlined in these paragraphs) may proceed. In general, SYSTEM RESET should be used with caution, for although it has no effect on the computer's memory and will not destroy the program, it resets all I/O operations including interrupts, disarms all interrupts, and may cause data to be lost.

Note that when the computer is halted (HLT or illegal instruction or MANUAL mode), [RP] will be the address of the current instruction plus two, and [TIR] will be the next instruction destined for execution.

Automatically Entering a Word into a Register

The ECS instruction offers the operator the opportunity of entering a word into a register under program control, without taking the computer out of AUTO mode. The instruction ECS is explained in detail in Section 6; in typical use the instruction will operate in a loop that expects the keys to contain information specifying performance of some special function by the operator, such as loading a magnetic tape, or information needed by the software, such as memory size. This loop would be exited when the position of the keys is changed and would respond to their new condition in a selective manner.

TAPE DRIVE OPERATING PROCEDURES

These paragraphs describe tape loading and unloading, and the functions of the 8512 controls and indicators. Tape threading techniques are easily mastered. The only precaution to be observed concerns the handling of tape reels rather than the unit itself — avoid exerting pressure on the reel flanges. They are a relatively flexible material; squeezing them together can exert pressure on the tape edges. This can damage the tape and might result in dropouts and other errors. Therefore, handle tape reels with care.

Loading Tape

Examine the tape threading path shown in Figure 9-2. Not shown is the beige plastic protective cover over the head

assembly and associated components. The protective cover may be removed by gently pulling it straight out from the tape deck (it mounts on two friction-loaded pins). Tape may be loaded with this cover in place; hence, its removal is not recommended.

a. Press the toggle in the center of the left-hand spindle at the point marked "PRESS". The opposite end will pivot forward and so remain.

b. Place the reel of tape on the spindle with the write-enable ring toward the tape deck. Ensure that it is completely seated, and press the toggle on the end opposite "PRESS". The reel is now locked in place.

c. Thread the tape around the movable guides and the capstan, and over the fixed guides and the read/write head assembly exactly as shown in the illustration. There is a spring-loaded black plastic pressure pad over the head which may be temporarily lifted. The tape must pass between it and the head.

d. Wrap the tape around the take-up reel in the direction shown. Hold it in place with the forefinger. With the other hand, turn the take-up reel counterclockwise until the end of the tape is overlapped and secured by the next layer. Close the transparent protective cover.

e. Press POWER switch on control panel and then press LOAD switch. They will light, and the buffer arms carrying the movable guides will move to their normal positions, approximately centered in their slots. The tape will load onto the tape-up reel until the load point is sensed. The LOAD indicator will then light. Pressing the ON LINE switch will place the unit on line, and System IV/70 will take control; no further action is needed.

Unloading Tape

Normally, System IV/70 will command the tape to rewind at the end of a tape operation. A rewound tape is one that is almost completely contained on the supply reel, but is still attached to the take-up reel. If it is desired to unload a tape in this state, ignore step *a* and proceed directly to step *b*. The unit must be off line to unload or rewind tape. If necessary press RESET switch to take unit off line.

a. Press the REWIND switch on the control panel. The tape will rewind until the beginning-of-tape marker is reached. It will then reverse direction and run briefly until load point is sensed.

b. Press REWIND switch. The tape will be completely returned to the supply reel.

c. Press the toggle in the center of the left-hand spindle at the point marked "PRESS". The supply reel unlocks and may be removed.

Controls and Indicators

- POWER** A combination pushbutton switch and indicator. It controls AC power to the unit and illuminates when power is on.
- LOAD** A combination pushbutton switch and indicator. Pressing it after threading tape causes tape to load until load point is sensed. It will remain illuminated as long as tape is at load point.
- ON LINE** A combination pushbutton switch and indicator. It is illuminated when the tape unit is under control of System IV/70. If the unit is off-line, pressing the switch will place it on-line.
- FILE PROTECT** An indicator light which illuminates when the write-enable ring is not attached to the supply reel. If the ring is not attached, a

write operation cannot be performed, and data on the tape cannot be erased.

- REWIND** A pushbutton switch that functions only when tape unit is off-line. It causes high-speed reverse tape motion. Rewind can be stopped by pressing RESET; otherwise, tape will stop at load point.
- FORWARD** A combination pushbutton switch and indicator that functions only when tape unit is off-line. It causes tape to move forward at normal speed until RESET is pressed.
- REVERSE** A combination pushbutton switch and indicator whose function is identical to REWIND, except speed is normal.
- RESET** A pushbutton switch that stops tape motion regardless of the command that started it and regardless of on-line/off-line condition. If pressed when unit is on-line, it will take unit off-line and turn off ON LINE indicator.

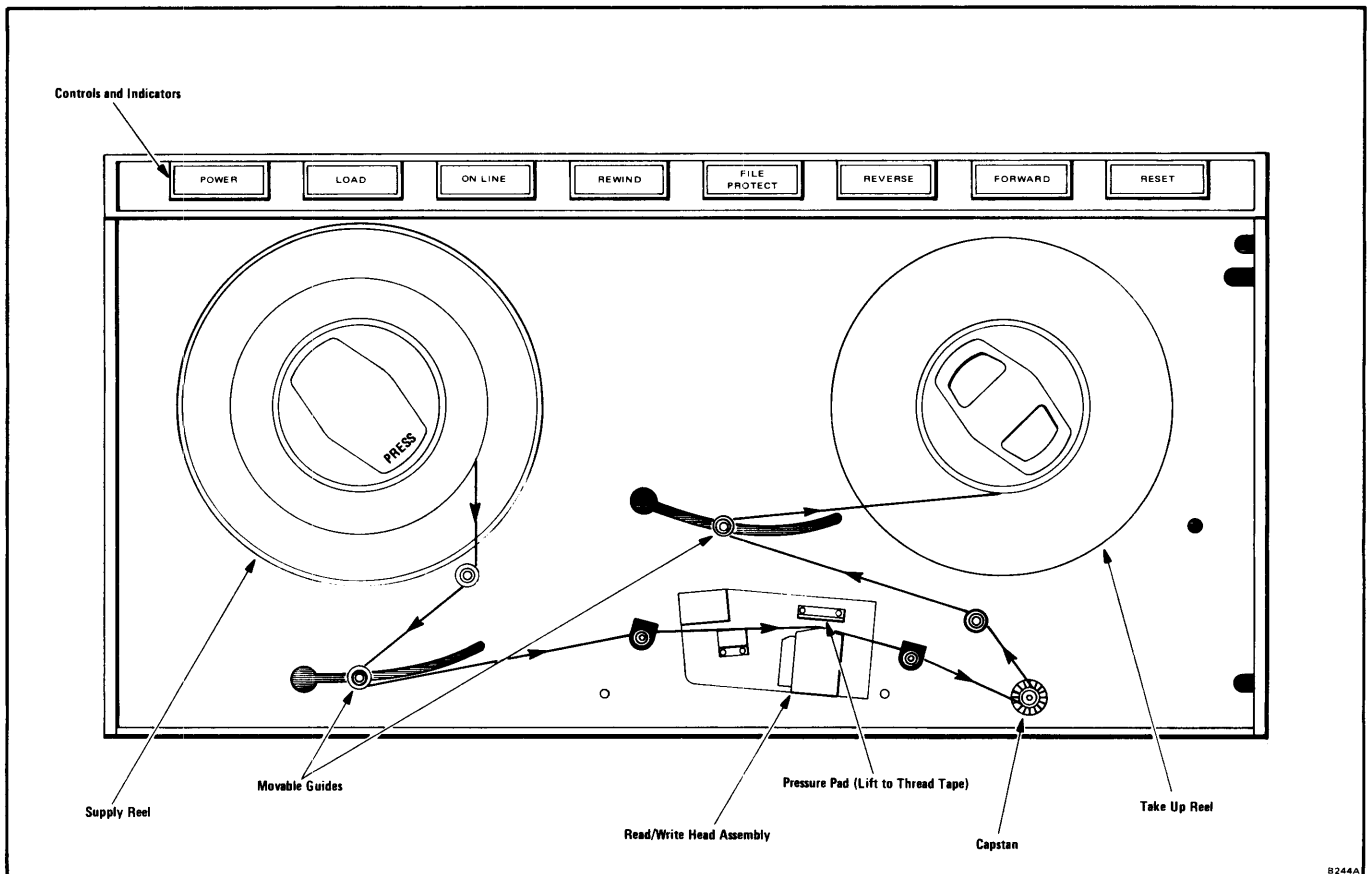


Figure 9-2. Magnetic Tape Drive Controls and Indicators

Appendix A

System IV/70 Character Set

Octal Code	Keyboard Character	Control Character Interpretation (ASCII)	Display Character	Display Character Meaning
000	= ^c	NUL Null	•	Dot
001	A ^c	SOH Start of Heading (CC)	△ ①	
002	B ^c	STX Start of Text (CC)	ⓑ ①	
003	C ^c	ETX End of Text (CC)	¢	Cent Sign
004	D ^c	EOT End of Transmission (CC)	▲	New Line Symbol
005	E ^c	ENQ Enquiry (CC)	≠ ① ②	
006	F ^c	ACK Acknowledge (CC)	+ ①	
007	G ^c	BEL Bell	▣ ①	
010	H ^c	BS Backspace (FE)	# ① ②	
011	I ^c	HT Horizontal Tabulation (FE)	←	Back Arrow
012	J ^c	LF Line Feed (FE)	\	Left Diagonal Graphic
013	K ^c	VT Vertical Tabulation (FE)	/	Right Diagonal Graphic
014	L ^c	FF Form Feed (FE)	£	British Pounds
015	M ^c	CR Carriage Return (FE)	■	Check Symbol
016	N ^c	SO Shift Out	¬	Logical Not
017	O ^c	SI Shift In		Left Vertical Graphic
020	P ^c	DLE Data Link Escape (CC)		Right Vertical Graphic
021	Q ^c	DC1 Device Control 1	[Opening Bracket
022	R ^c	DC2 Device Control 2	\	Check Symbol
023	S ^c	DC3 Device Control 3	-	Destructive Cursor or End of Message Symbol
024	T ^c	DC4 Device Control 4	√ ①	
025	U ^c	NAK Negative Acknowledge (CC)	-	Overline Graphic
026	V ^c	SYN Synchronous Idle (CC)	^	Circumflex
027	W ^c	ETB End of Transmission Block (CC)]	Closing Bracket
030	X ^c	CAN Cancel	□	Frame Cursor
031	Y ^c	EM End of Medium	∨ ① ②	
032	Z ^c	SUB Substitute	■	Block Cursor
033	+ ^c	ESC Escape	°	Degrees Sign
034	, ^c (comma)	FS File Separator (IS)	◀	Stop Delta
035	_ ^c (minus)	GS Group Separator (IS)		Nondestructive Cursor
036	ˆ ^c	RS Record Separator (IS)	▶	Start Manual Input Symbol or Start of Message Symbol or Start Delta
037	/ ^c	US Unit Separator (IS)	\	Reverse Slash

^c CONTROL key pressed at same time (FE) Format Effector

(CC) Communication Control (IS) Information Separator

② Blank for 7002 Processing Unit.

① These symbols are currently displayed but not supported. Other symbols may be substituted on later models.

Octal Code	Keyboard Character	Display Character	Display Character Meaning
040	space		Space or blank
041	! ①	!	Exclamation point
042	" ①	"	Quotation marks
043	# ①	#	Number sign
044	\$ ①	\$	Dollar sign
045	% ①	%	Percent sign
046	& ①	&	Ampersand
047	' (7 ^S) ①	'	Apostrophe, prime, or closing single quotation mark
050	(①	(Opening parenthesis
051) ①)	Closing parenthesis
052	*	*	Asterisk
053	+	+	Plus sign
054	,	,	Comma
	(comma)		
055	-	-	Minus sign or hyphen (dash)
	(minus)		
056	.	.	Period or decimal point
057	/	/	Slash
060	0	0	
	(zero)		
061	1	1	
062	2	2	
063	3	3	
064	4	4	
065	5	5	
066	6	6	
067	7	7	
070	8	8	
071	9	9	
072	:	:	Colon
073	;	;	Semicolon
074	<	<	Less than sign
075	=	=	Equals sign
076	>	>	Greater than sign
077	?	?	Question mark

① Shifted numeric key only; not a shifted numeric data island key.
S SHIFT key pressed at same time

Octal Code	Keyboard Character	Display Character	Display Character Meaning
100	@ (- ^S)	@	Commercial at sign
101	A ^S	A	
102	B ^S	B	
103	C ^S	C	
104	D ^S	D	
105	E ^S	E	
106	F ^S	F	
107	G ^S	G	
110	H ^S	H	
111	I ^S	I	
112	J ^S	J	
113	K ^S	K	
114	L ^S	L	
115	M ^S	M	
116	N ^S	N	
117	O ^S	Ø	
120	P ^S	P	
121	Q ^S	Q	
122	R ^S	R	
123	S ^S	S	
124	T ^S	T	
125	U ^S	U	
126	V ^S	V	
127	W ^S	W	
130	X ^S	X	
131	Y ^S	Y	
132	Z ^S	Z	
133	÷	÷	Divide sign
134	×	×	Multiply sign
135	(- ^S)		Logical OR or centered vertical graphic
136	EXP ↑ (÷ ^S)	↑	Up arrow, exponent sign, or alternate cursor
137	— (X ^S)	—	Underline graphic, nondestructive or alternate cursor

S SHIFT key pressed at same time

Octal Code	Keyboard Character	Display Character	Display Character Meaning
140	' (0 ^S) ① (zero)		Grave accent or opening single quotation mark
141	A	a	
142	B	b	
143	C	c	
144	D	d	
145	E	e	
146	F	f	
147	G	g	
150	H	h	
151	I	i	
152	J	j	
153	K	k	
154	L	l	
155	M	m	
156	N	n	
157	O	o	
160	P	p	
161	Q	q	
162	R	r	
163	S	s	
164	T	t	
165	U	u	
166	V	v	
167	W	w	
170	X	x	
171	Y	y	
172	Z	z	
173	÷ ^C	{	Opening brace
174	× ^C		Centered stylized vertical line (distinguishable from logical OR)
175	* ^C	}	Closing brace
176	↑ ^C	~	Tilde
177	← ^C	//	Delete symbol

① Shifted numeric key only; not a shifted numeric data island key.
 C CONTROL key pressed at same time
 S SHIFT key pressed at same time

Octal Code	Keyboard Character	Conventional Keyboard Code Interpretation
200	↑	Cursor Up
201	←	Cursor Left
202	→	Cursor Right
203	↓	Cursor Down
204	EOM	End of Message
205	ATTEN	Attention
206	ROLL (↓ ^S)	Roll Down
207	ERASE (HOME ^S)	Erase Screen
210	HOME	Cursor Home
211	TAB	Horizontal Tab
212	ROLL (↑ ^S)	Roll Up
213	TAB ^S	Vertical Tab
214	EOM ^S	Shifted EOM
215	CURSOR RETURN	Cursor Return
216	CURSOR RETURN ^S	Shifted Cursor Return
217	INSERT (→ ^S)	Insert
220	DELETE (← ^S)	Delete
221	F1	Function Key 1
222	F2	Function Key 2
223	F3	Function Key 3
224	F4	Function Key 4
225	F5	Function Key 5
226	F6	Function Key 6
227	F7	Function Key 7
230	F8	Function Key 8
231	F9	Function Key 9
232	F10	Function Key 10
233	F11	Function Key 11
234	→ ^C	Control →
235	TOTAL	Total
236	↓ ^C	Control Roll Down
237	EOM ^C	Control EOM

C CONTROL key pressed at same time
 S SHIFT key pressed at same time

Octal Code	Keyboard Character	Conventional Keyboard Code Interpretation	
240	Not in Use		
241	Lightpen Character		
242	} Not in Use		
257			
260		0 ^C ①	Control 0
261		1 ^C ①	Control 1
262		2 ^C ①	Control 2
263	3 ^C ①	Control 3	
264	4 ^C ①	Control 4	
265	5 ^C ①	Control 5	
266	6 ^C ①	Control 6	
267	7 ^C ①	Control 7	
270	8 ^C ①	Control 8	
271	9 ^C ①	Control 9	
272	} Not in Use		
.			
.			
.			
277			
① Controlled numeric key only; not a controlled numeric data island key. C CONTROL key pressed at same time			

Octal Code	Keyboard Character	Conventional Keyboard Code Interpretation	
300	} Not in Use		
.			
.			
.			
.			
337	} Not in Use		
340			
.			
.			
.			
374	} HOME ^C	Control Home	
375		CURSOR RETURN ^C	Control Cursor Return
376		TAB ^C	Control TAB
377			
C CONTROL key pressed at same time			

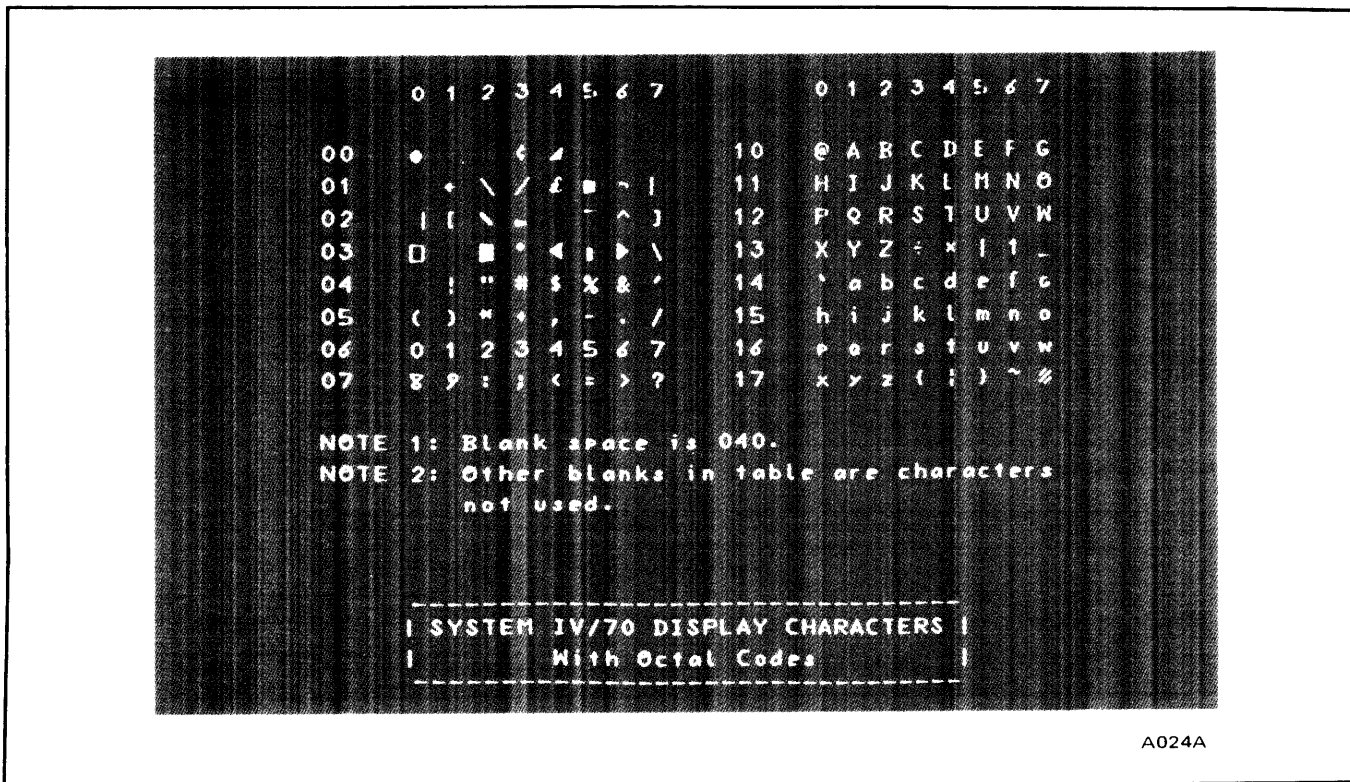


Figure A-1. System IV/70 Display Characters

Appendix B

Assembler Directives

ASSEMBLY FORMAT†	DESCRIPTION	PAGE
DATA CONTROL DIRECTIVES		
Whole Word Definition		
1 DCA 'Character String'	Define Constant in ASCII	8-6
1 DCN Expression	Define Constant Numeric (decimal or octal)	8-6
1 DCS Expression	Define Constant Single-precision floating	8-6
1 DCD Expression	Define Constant Double-precision floating	8-6
Part Word Definition		
1 PZE* Expression, x	Prefix 00 ₈	8-6
1 RPZE s,d,b,c	Prefix 007 ₈	8-7
1 MZE Expression	Prefix 777 ₈	8-7
Storage Allocation		
1 BSS Expression	Block Starting with Symbol	8-7
1 BES Expression	Block Ending with Symbol	8-7
Symbol Definition		
1 EQU Expression	Label Equals expression (label mandatory)	8-7
ASSEMBLER CONTROL DIRECTIVES		
Counter Control		
1 ORG Expression	Set l equal to \$, then set \$ equal to expression	8-7
Conditional Assembly		
L SKIP Expression	Skip assembly of val (expr) cards	8-8
L IFGT Expression, LABEL	Skip to LABEL on Greater than zero	8-8
L IFLT Expression, LABEL	Skip to LABEL on Less Than zero	8-8
L IFZO Expression, LABEL	Skip to LABEL on Zero	8-8
L IFNZ Expression, LABEL	Skip to LABEL on Non-Zero	8-8
Linkage Control		
ENTRY Symbol	Entry for a virtual symbol (label not legal)	8-8
1 EOP	Link to library and End Of Program	8-8
Miscellaneous		
END Expression	End of routine or program (label not legal)	8-9
FORCE 0 or 1	Force an even or odd location (label not legal)	8-9
† 1 = label. A label may be attached to any of the assembler directives preceded by l except for EQU where the label is mandatory. L is a label in the operand field of a conditional assembly statement.		

Appendix C

Machine Instructions

LISTED BY OPERATION CODES (x means indexing and/or indirect addressing; x < 7)

00x HLT	10x ORM	20x ANM	30x XOM	40x STZ
01x LDA1	11x INR	21x DEC	31x STA1	41x SAM
02x LD23	12x ADM	22x SKN	32x ST23	42x STP
03x LDA	13x ADA,ADD	23x SBA, SUB	33x CPA	43x STA
04x LDB	14x ORA,OR	24x ANA,AND	34x XOA,XOR	44x STB
05x LD1	15x AD1	25x SB1	35x CP1	45x ST1
06x LD2	16x AD2	26x SB2	36x CP2	46x ST2
07x LD3	17x AD3	27x SB3	37x CP3	47x ST3
50x SLR	60x SKZ	70x XEC	007 LCL	107 ROR
51x SLRD	61x MCC	71x BRM	017 LPL	117 RADD,RCC
52x SLA	62x BOF	72x BRA	027 RCL	127 MPY
53x SLAD	63x BZO	73x BNZ	037 RLC	137 MVL
54x SRL	64x BMI	74x BPL	047 LCR	147 UFA
55x SRLD	65x BCR	75x BC1	057 LPR	157 FAD
56x SRA	66x BAL	76x BC2	067 RCR,RCPY,NOP	167 FMP
57x SRAD	67x BGT	77x BC3	077 RRC	177 MVR
207 RAND	307 RXOR	407 SCL	507 BRD	607 CPN
217 RSUB	317 RCM2	417 SPL	517 BRR	617 MVCR
227 DIV	327 POP	427 PUSH	527 EXCT	627 DADD
237 MVE	337 UP	437 DOWN	537 EXSN	637 CPL
247 CDA2	347 IN	447 SCR	547 PIA	647 DSUB
257 FSB	357 TRT	457 SPR	557 PID	657 ODD
267 FDV	367 †	467 TRAP	567 PIR	667 MVCL
277 IOB	377 BOOT	477 ECS	577 IOID	677 IO

† Unpredictable results will be obtained if this op code is executed.


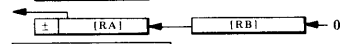
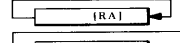
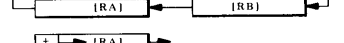
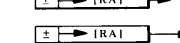
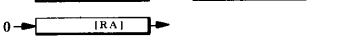
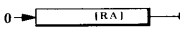
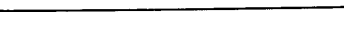
LISTED BY MNEMONICS AND FUNCTIONAL GROUPS (see page C-2)

ADA 13x E	BRD 507 I	FMP 167 F	MVCL 667 A	RCR 067 J	SRA 56x H
ADD 13x E	BRM 71x I	FSB 257 F	MVCR 617 A	RLC 037 J	SRAD 57x H
ADM 12x E	BRR 517 I	HLT 00x L	MVE 237 B	ROR 107 J	SRL 54x H
AD1 15x E	BZO 63x I	IN 347 C	MVL 137 B	RRC 077 J	SRLD 55x H
AD2 16x E	CDA2 247 J	INR 11x I	MVR 177 B	RSUB 217 J	STA 43x D
AD3 17x E	CPA 33x G	IO 677 N	NOP 067 L	RXOR 307 J	STA1 31x D
ANA 24x K	CPL 637 A	IOB 277 N	ODD 657 L	SAM 41x D	STB 44x D
AND 24x K	CPN 607 A	IOID 577 M	OR 14x K	SBA 23x E	STP 42x D
ANM 20x K	CP1 35x G	LCL 007 B	ORA 14x K	SB1 25x E	STZ 40x D
BAL 66x I	CP2 36x G	LCR 047 B	ORM 10x K	SB2 26x E	ST1 45x D
BCR 65x I	CP3 37x G	LDA 03x D	PIA 547 M	SB3 27x E	ST2 46x D
BC1 75x I	DADD 627 A	LDA1 01x D	PID 557 M	SCL 407 B	ST3 47x D
BC2 76x I	DEC 21x I	LDB 04x D	PIR 567 M	SCR 447 B	ST23 32x D
BC3 77x I	DIV 227 E	LD1 05x D	POP 327 C	SKN 22x I	SUB 23x E
BGT 67x I	DOWN 437 C	LD2 06x D	PUSH 427 C	SKZ 60x I	TRAP 467 L
BMI 64x I	DSUB 647 A	LD3 07x D	RADD 117 J	SLA 52x H	TRT 357 B
BNZ 73x I	ECS 477 N	LD23 02x D	RAND 207 J	SLAD 53x H	UFA 147 F
BOF 62x I	EXCT 527 N	LPL 017 B	RCC 117 L	SLR 50x H	UP 337 C
BOOT 377 N	EXSN 537 N	LPR 057 B	RCL 027 J	SLRD 51x H	XEC 70x L
BPL 74x I	FAD 157 F	MCC 61x L	RCM2 317 J	SPL 417 B	XOA 34x K
BRA 72x I	FDV 267 F	MPY 127 E	RCPY 067 J	SPR 457 B	XOM 30x K
					XOR 34x K

LISTED BY FUNCTIONS

Assembler Format ①	Octal Code	Condition Codes	Name	Timing ②	Equation	Page
A DECIMAL OPTION						
1 CPL sbs,sbd,L	637	ZMC	Compare block Logic	14+6Q	[D Memory Block] : [S Memory Block] ; bit-by-bit.	5-16
1 CPN sbs,sbd,L1,L2	607	OZMC	Compare block Numeric	16+6P	[D Memory Block] : [S Memory Block] ; ASCII	5-16
1 DADD sbs,sbd,L1,L2	627	OZMC	Decimal Addition	18+8P (+6+6P if recompl)	[D Memory Block] + [S Memory Block] → [D Memory Block] ; ASCII.	5-15
1 DSUB sbs,sbd,L1,L2	647	OZMC	Decimal Subtraction	18+8P (+6+6P if recompl)	[D Memory Block] - [S Memory Block] → [D Memory Block] ; ASCII.	5-16
1 MVCR sbs,sbd,L	617	--	Move Character Right	20+4Q	[S Memory Block] → [D Memory Block] , left to right	5-17
1 MVCL sbs,sbd,L	667	--	Move Character Left	20+4Q	[S Memory Block] → [D Memory Block] , right to left	5-17
B WORD- AND CHARACTER-MANIPULATION						
1 LCL e	007	Z	Load Character Left	22,32,32	See Text, Section 5	5-6
1 LCR e	047	Z	Load Character Right	16,32,36	See Text, Section 5	5-5
1 LPL e	017	Z	Load Parallel Left	42,42,26	See Text, Section 5	5-7
1 LPR e	057	Z	Load Parallel Right	20,38,42	See Text, Section 5	5-7
1 MVE c	237	--	Move block	6+4W	See Text, Section 5	5-1
1 MVL b,c	137	--	Move block Left	2+28W	See Text, Section 5	5-2
1 MVR b,c	177	--	Move block Right	2+28W	See Text, Section 5	5-1
1 SCL e	407	Z	Store Character Left	24,38,38	See Text, Section 5	5-6
1 SCR e	447	Z	Store Character Right	20,38,42	See Text, Section 5	5-6
1 SPL e	417	Z	Store Parallel Left	14	See Text, Section 5	5-8
1 SPR e	457	Z	Store Parallel Right	14	See Text, Section 5	5-8
1 TRT e	357	--	Translate bytes	144 max	See Text, Section 5	5-8
C LIST PROCESSING						
1 DOWN e	437	--	Down list	12	See Text, Section 5	5-13
1 IN e	347	--	Insert into list	20, 16ns	See Text, Section 5	5-13
1 POP e	327	--	Pop up list	16	See Text, Section 5	5-11
1 PUSH e	427	--	Push down list	22, 18ns	See Text, Section 5	5-11
1 UP e	337	--	Up list	12, 10ns	See Text, Section 5	5-11
D LOAD/STORE						
1 LDA* e,x	03x	--	Load RA	6	[EA] → [RA]	4-1
1 LDA1* e,x	01x	--	Load RA & X1	12	[EA] → [RA]; [EA ∪ 1] → [X1]	4-1
1 LDB* e,x	04x	--	Load RB	6	[EA] → [RB]	4-1
1 LD1* e,x	05x	--	Load X1	6	[EA] → [X1]	4-1
1 LD2* e,x	06x	--	Load X2	6	[EA] → [X2]	4-1
1 LD3* e,x	07x	--	Load X3	6	[EA] → [X3]	4-1
1 LD23* e,x	02x	--	Load X2 & X3	12	[EA] → [X2]; [EA ∪ 1] → [X3]	4-1
1 SAM* e,x	41x	--	Store RA address	12	[RA] ₉₋₂₃ → [EA] ₉₋₂₃ ; [EA] ₀₋₈ → [RA] ₀₋₈	4-2
1 STA* e,x	43x	--	Store RA	8	[RA] → [EA]	4-2
1 STA1* e,x	31x	--	Store RA & X1	12	[RA] → [EA]; [X1] → [EA ∪ 1]	4-2
1 STB* e,x	44x	--	Store RB	8	[RB] → [EA]	4-2
1 STP* e,x	42x	--	Store RP	8	[RP] → [EA]	4-2
1 STZ* e,x	40x	--	Store Zero	8	[R0] → [EA]	4-2
1 ST1* e,x	45x	--	Store X1	8	[X1] → [EA]	4-2
1 ST2* e,x	46x	--	Store X2	8	[X2] → [EA]	4-2
1 ST3* e,x	47x	--	Store X3	8	[X3] → [EA]	4-2
1 ST23* e,x	32x	--	Store X2 & X3	12	[X2] → [EA]; [X3] → [EA ∪ 1]	4-3

A027C

Assembler Format ①	Octal Code	Condition Codes	Name	Timing ②	Equation	Page
E FIXED POINT						
1 ADA* e,x	13x	OZMC	Add to RA	8	$[EA] + [RA] \rightarrow [RA]$	4-3
1 ADM* e,x	12x	OZMC	Add to Memory	10	$[RA] + [EA] \rightarrow [EA]$	4-4
1 AD1* e,x	15x	OZMC	Add to X1	8	$[EA] + [X1] \rightarrow [X1]$	4-3
1 AD2* e,x	16x	OZMC	Add to X2	8	$[EA] + [X2] \rightarrow [X2]$	4-3
1 AD3* e,x	17x	OZMC	Add to X3	8	$[EA] + [X3] \rightarrow [X3]$	4-3
1 DIV c	227	ZMC	Divide	18+8N	$[RA-RB] \div [X2] \rightarrow [RA]$, scaled right; remainder $\rightarrow [RB]$, scaled left	4-4
1 MPY c	127	ZMC	Multiply	24+8N	$[RA] \times [X2] \rightarrow [RA-RB]$	4-5
1 SBA* e,x	23x	OZMC	Subtract from RA	8	$[RA] - [EA] \rightarrow [RA]$	4-4
1 SB1* e,x	25x	OZMC	Subtract from X1	8	$[X1] - [EA] \rightarrow [X1]$	4-4
1 SB2* e,x	26x	OZMC	Subtract from X2	8	$[X2] - [EA] \rightarrow [X2]$	4-4
1 SB3* e,x	27x	OZMC	Subtract from X3	8	$[X3] - [EA] \rightarrow [X3]$	4-4
F FLOATING POINT						
1 FAD	157	ZMC	Floating Add	46+4N ₁ +8N ₂	$[RA,X1] + [X2,X3] \rightarrow [RA,X1]$. If $[X1] > [X3]$, $[RA] \rightarrow [X2]$	4-6
1 FDV	267	ZMC	Floating Divide	228	$[RA,X1] \div [X2,X3] \rightarrow [RA,X1]$. Remainder $\rightarrow [RB]$	4-7
1 FMP	167	ZMC	Floating Multiply	220+8N ₃	$[RA,X1] \times [X2,X3] \rightarrow [RA-RB,X1]$	4-7
1 FSB	257	ZMC	Floating Subtract	52+4N ₁ +8N ₂	$[RA,X1] - [X2,X3] \rightarrow [RA,X1]$. If $[X1] > [X3]$, $[RA] \rightarrow [X2]$.	4-7
1 UFA	147	ZMC	Unnormalized Floating Add	90	Same as FAD	4-7
G COMPARISON						
1 CPA* e,x	33x	OZMC	Compare RA	8	$[RA] : [EA]$, set CC.	4-9
1 CP1* e,x	35x	OZMC	Compare X1	8	$[X1] : [EA]$, set CC.	4-9
1 CP2* e,x	36x	OZMC	Compare X2	8	$[X2] : [EA]$, set CC.	4-9
1 CP3* e,x	37x	OZMC	Compare X3	8	$[X3] : [EA]$, set CC.	4-9
H SHIFT ACCUMULATOR						
1 SLA* e,x	52x	O	Left Arithmetic single	9+3K (+1)		4-10
1 SLAD* e,x	53x	O	Left Arithmetic Double	8+4K		4-11
1 SLR* e,x	50x	--	Left Rotate single	6+2K		4-11
1 SLRD* e,x	51x	--	Left Rotate Double	9+5K (+1)		4-12
1 SRA* e,x	56x	--	Right Arithmetic single	6+2K		4-10
1 SRAD* e,x	57x	--	Right Arithmetic Double	8+4K		4-11
1 SRL* e,x	54x	--	Right Logical single	6+2K		4-11
1 SRLD* e,x	55x	--	Right Logical Double	8+4K		4-11
I BRANCH/SKIP						
1 BAL* e,x	66x	--	Branch & Link	6	$[RP] \rightarrow [X2]; EA \rightarrow [RP]$	4-13
1 BCR* e,x	65x	--	Branch if Carry	6	If C = 1, $EA \rightarrow [RP]$	4-15
1 BC1* e,x	75x	--	Branch & Count X1	10	$[X1] + 1 \rightarrow [X1]$. If $[X1] \neq 0$, $EA \rightarrow [RP]$	4-16
1 BC2* e,x	76x	--	Branch & Count X2	10	$[X2] + 1 \rightarrow [X2]$. If $[X2] \neq 0$, $EA \rightarrow [RP]$	4-16
1 BC3* e,x	77x	--	Branch & Count X3	10	$[X3] + 1 \rightarrow [X3]$. If $[X3] \neq 0$, $EA \rightarrow [RP]$	4-16
1 BGT* e,x	67x	--	Branch if logically Greater	6	If $\bar{Z} \cap \bar{C} = 1$, $EA \rightarrow [RP]$	4-15

A028C

Assembler Format ①	Octal Code	Condition Codes	Name	Timing ②	Equation	Page
I BRANCH/SKIP						
1 BMI* e,x	64x	--	Branch on Minus	6	If M = 1, EA → [RP]	4-14
1 BNZ* e,x	73x	--	Branch on Nonzero	6	If Z = 0, EA → [RP]	4-15
1 BOF* e,x	62x	O	Branch on Overflow	6	If O = 1, EA → [RP]; 0 → 0	4-15
1 BPL* e,x	74x	--	Branch on not minus	6	If M = 0, EA → [RP]	4-14
1 BRA* e,x	72x	--	Branch unconditional	6	EA → [RP]	4-12
1 BRD e	507	OZMC	Branch Return Debreak	8	[EA] ₂₋₅ → CC; [EA] → [RP]; issue debreak	4-14
1 BRM* e,x	71x	--	Branch & Mark	10+I	[RP] ₉₋₂₃ → [EA] ₉₋₂₃ ; status bits → [EA] ₀₋₅ ; EA + 1 → [RP] ₉₋₂₃	4-13
1 BRR e	517	OZMC	Branch Return	8	[EA] ₂₋₅ → CC; [EA] → [RP]	4-14
1 BZO* e,x	63x	--	Branch on Zero	6	If Z = 1, EA → [RP]	4-14
1 DEC* e,x	21x	--	Decrement memory, skip if zero	14	[EA] - 1 → [EA]. If [EA] = 0, [RP] + 1 → [RP]	4-17
1 INR* e,x	11x	--	Increment memory, skip if zero	10+I	See Text, Section 4	4-16
1 SKN* e,x	22x	--	Test memory, Skip if Negative	8	If [EA] < 0, [RP] + 1 → [RP]	4-17
1 SKZ* e,x	60x	--	Test memory, Skip if Zero	10	If [EA] = 0, [RP] + 1 → [RP]	4-17
J REGISTER-TO-REGISTER						
1 CDA2	247	--	Copy Double	8	[RA] → [X2]; [X1] → [X3]	4-18
1 RADD s,d,b	117	OZMC	Register Add	8	[S] + [D] → [D], selected bytes	4-18
1 RAND s,d,b	207	ZM	AND source to dest	8	[S] ∩ [D] → [D], selected bytes	4-19
1 RCL s,d,b,c	027	--	Copy then Rotate Left	6+2K	[S] → [D], selected bytes; rotate [D] left [C] locations	4-20
1 RCM2 s,d	317	OZMC	2's Complement	10	- [S] → [D]	4-20
1 RCPY s,d,b	067	--	Copy source to dest	6	[S] → [D], selected bytes	4-18
1 RCR s,d,b,c	067	--	Copy then Rotate Right	6+2K	[S] → [D], selected bytes; rotate [D] right [C] locations	4-20
1 RLC s,d,b,c	037	--	Rotate Left, then Copy	8+2K	Rotate [S] left [C] locations; rotated quantity → [D], selected bytes; [S] unchanged.	4-21
1 ROR s,d,b	107	ZM	OR source to dest	8	[S] ∪ [D] → [D], selected bytes	4-19
1 RRC s,d,b,c	077	--	Rotate Right, then Copy	8+2K	Rotate [S] right [C] locations; rotated quantity → [D], selected bytes; [S] unchanged.	4-21
1 RSUB s,d,b	217	OZMC	Register Subtract	8	[D] - [S] → [D], selected bytes	4-18
1 RXOR s,d,b	307	ZM	XOR source to dest	8	[S] ∪ [D] → [D], selected bytes	4-19
K LOGICAL						
1 ANA* e,x	24x	ZM	AND to RA	8	[EA] ∩ [RA] → [RA]	4-21
1 ANM* e,x	20x	ZM	AND to Memory	10	[RA] ∩ [EA] → [EA]	4-22
1 ORA* e,x	14x	ZM	OR to RA	8	[EA] ∪ [RA] → [RA]	4-22
1 ORM* e,x	10x	ZM	OR to Memory	10	[RA] ∪ [EA] → [EA]	4-22
1 XOA* e,x	34x	ZM	XOR to RA	8	[EA] ∪ [RA] → [RA]	4-22
1 XOM* e,x	30x	ZM	XOR to Memory	10	[RA] ∪ [EA] → [EA]	4-23
L CONTROL						
1 HLT e	00x	--	Halt	6	Not applicable	4-23
1 MCC* e,x	61x	ZM	Memory set CC	8	[R0] + [EA] → [EA], set CC	4-23
1 ODD s,d,b,c	657	--	Compute odd parity	10+3K (+1)	See Text, Section 4.	4-24
1 NOP	067	--	No operation	6	Not applicable	4-23
1 RCC s	117	ZM	Register set CC	8	[R0] + [S] → [S], set CC	4-23
1 TRAP e	467	--	Trap to 41 _s	6	[41 _s] → [TIR]	4-24
1 XEC* e,x	70x	--	Execute [EA]	2	[EA] → [TIR]	4-24

A029C

Assembler Format ①	Octal Code	Condition Codes	Name	Timing ②	Equation	Page
M INTERRUPT						
l IOID e	577	--	Indirect Interrupt	10	See Text, Section 7	7-3
l PIA e	547	--	Priority Interrupt Arm	6	See Text, Section 7	7-3
l PID e	557	--	Priority Interrupt Disarm	6	See Text, Section 7	7-3
l PIR e	567	--	Priority Interrupt Reset	8+6T ③	See Text, Section 7	7-3
N INPUT/OUTPUT						
l BOOT s,d	377	Z	Bootstrap load	NA	See Text, Section 6	6-3
l ECS d,b	477	--	Enter Console keys	8	[Keys] → [D]	6-5
l EXCT e	527	--	External Command	6	[EA] ₂₀₋₂₃ → [EXC] ₀₋₃	6-5
l EXSN e	537	--	External Sense	6	If ([EA] ₂₀ ∩ EXS ₀) ∪ ([EA] ₂₁ ∩ EXS ₁) ∪ ([EA] ₂₂ ∩ EXS ₂) ∪ ([EA] ₂₃ ∩ EXS ₃) = 1, [RP] + 1 → [RP]	6-5
l IO e	677	--	Input/Output words	28+I+6(W-1)	See Text, Section 6	6-2
l IOB e	277	Z	Input/Output Bytes	82+I	See Text, Section 6	6-3

A030C

① The assembly language form of the instruction is given for reference purposes. The mnemonic is given in capital letters, label and operand information, with a few exceptions, in lower case. An asterisk (*) attached to the mnemonic indicates that indirect addressing may be used with this instruction. The significance of the other entries in the first column is as follows:

l = label. A label may be attached to any machine instruction.

e = expression. It must be possible for the assembler to reduce a multiterm expression to an address or a count. If the expression is a single symbol virtual, it will be evaluated by the loader. e is not optional.

e,x = expression plus indexing. The expression is the same as above, but indexing may also be applied. Indexing is always optional.

s,d = source and destination registers. These are not options.

b = byte control. If no byte control is given by the programmer for instructions where it is indicated, the assembler will supply 7 (all bytes).

c = count. For the various count options, see the discussion of the specified instruction in Section 4 or 5. Any count entered will be treated modulo 64.

sbs,sbd = Starting Byte locations, Source and Destination. Values may be 0, 1, or 2, indicating the starting bytes for operations in a Decimal Option instruction.

L = Length of memory block in bytes for a Decimal Option instruction.

L1 = Difference in lengths of memory blocks in bytes for a Decimal Option instruction.

L2 = Length in bytes of the Source memory block for a Decimal Option instruction.

② Timing is given in machine cycles; the cycle time may be 1.9 or 2.03 microseconds. A cycle time of 2.03 microseconds is recommended for optimum video display. Indexing adds four cycles; indirect addressing adds two cycles. Other symbols are as follows:

K = shift count (range = 0-63)

W = word count = count field plus 1 (range = 1-64)

ns = no skip

N = count (range = 0-23)

N₁ = prealign (range 0-63, average 8-9)

N₂ = normalize (range = 0-23, average = 5-6)

N₃ = 0, 1, or 2

I = 0 normal, 4 if fetched at interrupt

(+1) = one is added if number of cycles is odd

P = Length of Destination in words, Maximum 22

Q = Length of Block in words, Maximum 86

T = Number of interrupts

③ 6 for 7001 Processing Unit

Appendix D

Powers of 2 and 8

$2^n \text{ \& } 8^m$	m	n	$2^{-n} \text{ \& } 8^{-m}$
1	0	0	1.0
2		1	0.5
4		2	0.25
8	1	3	0.125
16		4	0.062 5
32		5	0.031 25
64	2	6	0.015 625
128		7	0.007 812 5
256		8	0.003 906 25
512	3	9	0.001 953 125
1 024		10	0.000 976 562 5
2 048		11	0.000 488 281 25
4 096	4	12	0.000 244 140 625
8 192		13	0.000 122 070 312 5
16 384		14	0.000 061 035 156 25
32 768	5	15	0.000 030 517 578 125
65 536		16	0.000 015 258 789 062 5
131 072		17	0.000 007 629 394 531 25
262 144	6	18	0.000 003 814 697 265 625
524 288		19	0.000 001 907 348 632 812 5
1 048 576		20	0.000 000 953 674 316 406 25
2 097 152	7	21	0.000 000 476 837 158 203 125
4 194 304		22	0.000 000 238 418 579 101 562 5
8 388 608		23	0.000 000 119 209 289 550 781 25
16 777 216	8	24	0.000 000 059 604 644 775 390 625
33 554 432		25	0.000 000 029 802 322 387 695 312 5
67 108 864		26	0.000 000 014 901 161 193 847 656 25
134 217 728	9	27	0.000 000 007 450 580 596 923 828 125
268 435 456		28	0.000 000 003 725 290 298 461 914 062 5
536 870 912		29	0.000 000 001 862 645 149 230 957 031 25
1 073 741 824	10	30	0.000 000 000 931 322 574 615 478 515 625
2 147 483 648		31	0.000 000 000 465 661 287 307 739 257 812 5
4 294 967 296		32	0.000 000 000 232 830 643 653 869 628 906 25
8 589 934 592	11	33	0.000 000 000 116 415 321 826 934 814 453 125
17 179 869 184		34	0.000 000 000 058 207 660 913 467 407 226 562 5
34 359 738 368		35	0.000 000 000 029 103 830 456 733 703 613 281 25
68 719 476 736	12	36	0.000 000 000 014 551 915 228 366 851 806 640 625
137 438 953 472		37	0.000 000 000 007 275 957 614 183 425 903 320 312 5
274 877 906 944		38	0.000 000 000 003 637 978 807 091 712 951 660 156 25
549 755 813 888	13	39	0.000 000 000 001 818 989 403 545 856 475 830 078 125
1 099 511 627 776		40	0.000 000 000 000 909 494 701 772 928 237 915 039 062 5
2 199 023 255 552		41	0.000 000 000 000 454 747 350 886 464 118 957 519 531 25
4 398 046 511 104	14	42	0.000 000 000 000 227 373 675 443 232 059 478 759 765 625
8 796 093 022 208		43	0.000 000 000 000 113 686 837 721 616 029 739 379 882 812 5
17 592 186 044 416		44	0.000 000 000 000 056 843 418 860 808 014 869 689 941 406 25
35 184 372 088 832	15	45	0.000 000 000 000 028 421 709 430 404 007 434 844 970 703 125
70 368 744 177 664		46	0.000 000 000 000 014 210 854 715 202 003 717 422 485 351 562 5
140 737 488 355 328		47	0.000 000 000 000 007 105 427 357 601 001 858 711 242 675 781 25
281 474 976 710 656	16	48	0.000 000 000 000 003 552 713 678 800 500 929 355 621 337 890 625
562 949 953 421 312		49	0.000 000 000 000 001 776 356 839 400 250 464 677 810 668 945 312 5
1 125 899 906 842 624		50	0.000 000 000 000 000 888 178 419 700 125 232 338 905 334 472 656 25

USER'S COMMENTS

System IV/70 Computer Reference Manual

SIV/70-11-1C

Your comments will be considered for improving future documentation. Please give specific page and line references if appropriate.

- Which of the following best describes your occupation:
 - Programmer
 - Systems Analyst/Designer
 - Engineer
 - Operator
 - Instructor
 - Student
 - Manager
 - Customer Engineer
 - Other _____

- In what ways do you use this document?
 - Reference Manual
 - In a class
 - Self Study
 - Introduction to the Subject
 - Introduction to this System
 - Other _____

- Comments/Criticisms

Thank you for your assistance. No postage required if mailed in the USA.

fold

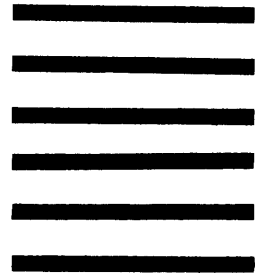
fold

FIRST CLASS
Permit No. 194
Cupertino,
California

BUSINESS REPLY MAIL
No Postage Necessary if Mailed in the United States

Postage will be Paid by . . .

FOUR-PHASE SYSTEMS
10420 North Tantau Ave.
Cupertino, Calif. 95014



Cut Along Here

Attention: Technical Publications

fold

fold

