Computer Museum

RTE Assembler

Reference Manual

HEWLETT hp PACKARD

HEWLETT-PACKARD COMPANY
11000 WOLFE ROAD, CUPERTINO, CALIFORNIA, 95014

Library Index No.
2RTE.320.92060-90005

PART NO. 92060-90005

CHANGE 2: 12/78
Printed in U.S.A. 6/78

PUBLICATION NOTICE

Information in this manual describes the RTE Assembler. Changes in text to document software updates subsequent to the initial release are supplied in manual update notices and/or complete revisions to the manual. The history of any changes to this edition of the manual is given below under "Publication History." The last change itemized reflects the software currently documented in the manual.

Any changed pages supplied in an update package are identified by a change number adjacent to the page number. Changed information is specifically identified by a vertical line (revision bar) on the outer margin of the page.

PUBLICATION HISTORY

Third Edition ..... Jun 78 (Software Rev. Code 1639)
Change 1 ..... Oct 78 (Software Rev. Code 1639)
Change 2 ..... Dec 78 (Software Rev. Code 1639)

NOTICE

The information contained in this document is subject to change without notice.

HEWLETT-PACKARD MAKES NO WARRANTY OF ANY KIND WITH REGARD TO THIS MATERIAL, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. Hewlett-Packard shall not be liable for errors contained herein or for incidental or consequential damages in connection with the furnishing, performance or use of this material.

Hewlett-Packard assumes no responsibility for the use or reliability of its software on equipment that is not furnished by Hewlett-Packard.

This document contains proprietary information which is protected by copyright. All rights are reserved. No part of this document may be photocopied, reproduced or translated to another program language without the prior written consent of Hewlett-Packard Company.

Copyright ©1978 by HEWLETT-PACKARD COMPANY

HP Computer Museum
www.hpmuseum.net

For research and education purposes only.

MANUAL CHANGE NOTICE

MANUAL IDENTIFICATION

Part Number: 92060-90005

Print Date: June 1978

Title: RTE Assembler Reference Manual

Library Index No: 2RTE.320.92060-90005

CHANGE IDENTIFICATION

Change Number: 2

Print Date: Dec 1978

Software Revision Code: 1639

THE PURPOSE OF THIS MANUAL CHANGE
is to accumulate all changes to the current edition of the manual. Earlier changes, if any, are contained herein for your convenience. *(If you have made all previous changes to this manual, you need only make the changes described under the change number indicated above.)*

CHANGED PAGES ARE IDENTIFIED
by the change number at the bottom of the page and a vertical line (change bar) in the outside margin to indicate the area of the text that has been changed..

NEW PAGES ARE IDENTIFIED
by the change number at the bottom of the page. "New" pages are those which were not present when the current edition of the manual was published.

TO UPDATE YOUR MANUAL
locate the Change Number, indicated above, on the back of this page and follow the instructions provided.

HEWLETT hp PACKARD

HEWLETT-PACKARD COMPANY
11000 WOLFE ROAD, CUPERTINO, CALIFORNIA, 95014

TECHNICAL MANUAL CHANGE
(92060-90005)

CHANGE 1:

Reason for Change 1: To correct information pertaining to ORG statement.

A. Title Page: Change print date in lower right corner to 10/78.

B. Page ii under "Third Edition" add:
Change 1 . . . . . Oct 78 (Software Rev. Code 1639)

C. Reader Comment Sheet: Change date to Oct 1978.

D. Replace pages 4-1 and 4-2 with included change pages.

CHANGE 2:

Reason for Change 2: To incorporate changes to the Formatter.

A. Replace existing pages with enclosed change pages.

PREFACE

This manual describes the Assembler which is designed to operate under control of the RTE Operating System.

This manual assumes that the reader is an experienced assembly language programmer who is familiar with operating systems.

The Assembler permits the programmer to use all supported machine instructions for the HP 21MX Computer and it is assumed that object programs produced by the Assembler will be executed on an HP 21MX Computer. However, the object program may be executed on other HP 2100 Series computers (2114, 2115, 2116, or 2100) if the following machine and pseudo instructions are *not* used:

Word Processing (described in paragraph 3-5)

Byte Processing (paragraph 3-6)

Bit Processing (paragraph 3-7)

Index Register Group (paragraph 3-11)

Floating Point (paragraph 3-18)

Dynamic Mapping System (paragraph 3-19)

DBL and DBR: Define Left Byte and Define Right Byte (paragraph 4-3)

BYT: Define Octal Byte Constants (paragraph 4-4)

MIC: Define User Instruction (paragraph 4-8)

Computer Museum

The Floating Point instructions (paragraph 3-18) may be used on a 2100 A/S with the 12901A Floating Point Option.

If the object programs produced by the Assembler are relocated and executed under control of an operating system other than the RTE Operating System, the following restrictions apply:

ENT pseudo instructions with absolute or common symbols as operands must *not* be used.

I/O instructions using externally-defined select codes must *not* be used.

I/O select codes must *not* be defined via the ENT pseudo instruction.

Memory reference instructions must *not* refer to *external* symbols with *offset* values.

When assembling programs to be run under control of the Basic Control System (BCS) (see the *Basic Control System Manual*, part no. 02116-90017), the following restrictions also apply:

Absolute operands greater than 77_8 are illegal in relocatable programs. However, such usage will *not* be diagnosed as errors by the loader; instead, it will result in errors during execution of the object program.

The content of this manual is as follows:

Section I	discusses the assembly process in general, program relocation, assembly options, and assembler input and output.
Section II	describes the source statement format.
Section III	describes all of the available machine instructions.
Section IV	describes all of the available assembler pseudo instructions.

In addition, nine appendices are supplied, as follows:

Appendix A	describes the Hewlett-Packard character set.
Appendix B	summarizes all of the available machine and pseudo instructions (including instruction formats).
Appendix C	presents a one-sentence definition of all available machine and pseudo instructions, arranged alphabetically by mnemonic.
Appendix D	presents a tabular summary of the binary format of all available machine instructions.
Appendix E	describes how to run an assembly.
Appendix F	describes use of the Formatter.
Appendix G	lists and describes all of the assembler error messages.
Appendix H	presents tape formats.
Appendix I	discusses the RTE Cross Reference Table Generator.

CONTENTS

Section I — Page

INTRODUCING THE ASSEMBLER

Assembly Processing 1-1
Symbolic Addressing 1-1
Memory Addressing 1-1
  Paging 1-1
  Indirect Addressing 1-2
  Program Relocation 1-2
Program Location Counter 1-3
Source Program 1-3
Assembly Options 1-3
Binary Output 1-3
Symbol Table 1-3
List Output 1-6

Section II — Page

SOURCE STATEMENT FORMAT

Statement of Characteristics 2-1
  Field Delimiters 2-1
  Character Set 2-1
  Statement Length 2-1
Label Field 2-1
  Label Symbol 2-1
  Asterisk 2-2
Opcode Field 2-2
Operand Field 2-2
  Symbolic Terms 2-2
  Numeric Terms 2-4
  Asterisk 2-4
  Expression Operators 2-4
  Evaluation of Expressions 2-4
  Expression Terms 2-4
  Absolute and Relocatable Expressions 2-4
    Absolute Expressions 2-4
    Relocatable Expressions 2-6
  Literals 2-6
  Indirect Addressing 2-6
  Clear Flag Indicator 2-7
Comments Field 2-7

Section III — Page

MACHINE INSTRUCTIONS

Memory Reference 3-1
  Jump and Increment-Skip 3-1
  Add, Load and Store 3-1
  Logical Operations 3-2
  Word Processing (21MX Series Only) 3-2
  Byte Processing (21MX Series Only) 3-2
  Bit Processing (21MX Series Only) 3-3
Register Reference 3-4
  Shift-Rotate Group 3-4
  Alter-Skip Group 3-4
  Index Register Group (21MX Series Only) 3-5
  No-Operation Instruction 3-7
Input/Output, Overflow, and Halt 3-7
  Input/Output 3-8
  Overflow 3-9
  Halt 3-9
Extended Arithmetic Unit (EAU) 3-9
Floating Point 3-10
Dynamic Mapping System (21MX Series Only) 3-10
  Memory Addressing 3-11
  Status and Violation Registers 3-11
  Map Segmentation 3-11
  Power Fail Characteristics 3-11
  Protected Mode 3-12
  MEM Violation 3-12
  Dynamic Mapping System Instructions 3-12
HP 21MX Series Fences 3-18

Section IV — Page

PSEUDO INSTRUCTIONS

Assembler Control 4-1
Object Program Linkage 4-5
Address and Symbol Definition 4-11
Constant Definition 4-14
Storage Allocation 4-19
Assembly Listing Control 4-19
Arithmetic Subroutine Calls 4-20
Define User Instruction (21MX Series Only) 4-21
  "Jump to Microprogram" 4-21
    Example 4-21
  Combining Multiple Mnemonics 4-21
    Example 4-21
  Defining Constants 4-21
    Example 4-22

Appendix A — Page

HP CHARACTER SET A-1

Appendix B — Page

SUMMARY OF INSTRUCTIONS

Machine Instructions B-2
  Memory Reference B-2
    Jump and Increment-Skip B-2
    Add, Load and Store B-2
    Logical B-2
    Word Processing B-2
    Byte Processing B-3
    Bit Processing B-3
  Register Reference B-3
    Shift-Rotate B-3
    No-Operation B-4
    Alter-Skip B-4
    Index Register B-5
  Input/Output, Overflow, and Halt B-6
    Input/Output B-6
    Overflow B-6
    Halt B-6
  Extended Arithmetic Unit B-6
  Floating Point B-7

CONTENTS (continued)

Memory Expansion B-7
Pseudo Instructions B-9
Assembler Control B-9
Object Program Linkage B-9
Address and Symbol Definition B-9
Constant Definition B-10
Storage Allocation B-10
Assembly Listing Control B-10
Define User Instruction B-10

Appendix C Page
ALPHABETIC LIST OF INSTRUCTIONS C-1

Appendix D Page
CONSOLIDATED CODING SHEETS D-1

Appendix E Page
RUNNING ASSEMBLIES
Assembler I/O E-1
Assembler Operation E-1
Messages During Assembly E-2

Appendix F Page
THE FORMATTER
Input and Output F-1
Records F-1
Formatted Input/Output F-1
Format Specifications F-4
Conversion Specifications F-4
Editing Specifications F-4
E Specification F-4
Output F-4
Input F-5
Rules for E Field Input F-5
F Specification F-5
Output F-5
Input F-5
I Specification F-5
Output F-5
Input F-6
O, K, and @ Specifications F-6
Output F-6
Input F-6
A and R Specifications F-6
Output F-6
Input F-7
X Specification F-7
Output F-7
Input F-7
H and " " Specifications (Literal Strings) F-7
Output F-7
Input F-8
/Specification F-8
How to Put Formats Together F-8
Free-Field Input F-9
Data Item Delimiters F-9
Floating-Point Input F-9
Octal Input F-9
Record Terminator F-10
Comments Within Input F-10
Example Calling Sequences F-10
Internal Conversion F-10
Buffered I/O with the Formatter F-11

Appendix G Page
ASSEMBLER ERROR MESSAGES G-1

Appendix H Page
TAPE FORMATS
NAM Record H-1
ENT Record H-2
EXT Record H-3
DBL Record H-4
END Record H-5
Absolute Tape Format H-6

Appendix I Page
RTE CROSS REFERENCE TABLE GENERATOR
Computer Configuration I-1
Functional and Operational Characteristics I-1
Output Format I-1
Pseudo Processing I-1
Double Defined Processing I-1
Undefined Label Processing I-1
Unused Label Processing I-2
Literal Processing I-2
Operation Directive I-2
Bounds I-2
Sample Cross-Reference Generation I-3

ILLUSTRATIONS

Title	Page
Source Program	1-4
Symbol Table Listing	1-5
Label Examples	2-3
Label Usage Examples	2-3
Symbolic Operand Examples	2-5
Expression Operator Examples	2-5
Indirect Addressing Example	2-7
Clear Flag Examples	2-7
Basic Memory Addressing Scheme	3-11
Expanded Memory Addressing Scheme	3-11
Map Segmentation	3-12
ORB Example	4-2
ORR Example (with Single ORG)	4-3
ORR Example (with Multiple ORG's)	4-3
IFN/XIF and IFZ/XIF Example	4-4
IFZ/XIF Example	4-4
COM Examples	4-6
ENT/EXT Examples	4-7
EXT with Offset	4-8
ENT in COMmon and ENT Defining An External I/O Reference	4-8
EXT, ENT for I/O Channel	4-9
Label RPL Octal Value	4-10
DEF Examples	4-11
Example of Incorrect Address Modification	4-11
Loader-Assigned Locations for Figure 4-8	4-12
Example of Correct Address Modification	4-12
Loader-Assigned Locations for Figure 4-10	4-12
ABS Examples	4-13
EQU Example	4-13
EQU Examples	4-14
ASC Example	4-15
DEC Examples (Integer)	4-16
DEC Examples (Floating Point)	4-16
DEC Examples (Floating Point)	4-16
DEX Memory Format	4-17
DEX Examples	4-17
OCT Examples	4-18
BYT Examples	4-19
Input Calling Sequence Selection	F-2
Output Calling Sequence Selection	F-3
Buffered I/O with the Formatter	F-12

TABLES

Title	Page
Logical Memory Addresses/Pages	1-2
Control Statement Parameters	1-5
MEM Status Register Format	3-11
MEM Violation Register Format	3-12
Base Set Instruction Codes in Binary	D-2
Extended Instruction Group Codes in Binary	D-3

INTRODUCING THE ASSEMBLER

SECTION I

The Assembler translates symbolic source language instructions into an object program for execution on the computer. The source language provides mnemonic machine operation codes, assembler-directing pseudo instructions, and symbolic addressing. The assembled program may be absolute or relocatable.

The source program may be assembled as a complete entity or it may be subdivided into several relocatable subprograms (or a main program and several subroutines), each of which may be assembled separately. When relocatable object programs and subprograms are desired to be executed, they are relocated and linked to one another by the relocating loader.

Absolute object programs may be loaded by the Basic Binary Loader or the Basic Binary Disc Loader. There are no intermediate steps needed to prepare the code before it is executed.

The Assembler can read the source input from paper tape, punched cards, magnetic tape or the LS Area of the disc. The Assembler outputs the resultant object program on the standard punch output device and/or to the LG Area of the disc in a format acceptable to the RTE Relocating Loader.

1-1. ASSEMBLY PROCESSING

The Assembler is a two pass system. A *pass* is defined as a processing cycle of the source program input.

In the first pass, the Assembler creates a symbol table from the names used in the source statements and (if requested) prints a symbol table listing on the standard list output device. It also checks for certain possible error conditions and prints error messages on the console device if necessary.

During pass two, the Assembler again examines each statement in the source program along with the symbol table and produces the binary object program. It outputs the object program to the standard punch output device and/or to the LG Area of the disc. If requested, the Assembler also prints the source program listing on the standard list output device. Additional error messages may also be printed on the console device.

If the source input is being read from a non-disc device, it is written on the disc at the start of pass 1; for pass 2, the source is then read from the disc. However, if there is not sufficient space available on the disc to do this, the source input will have to be read through the non-disc device at the start of pass 2. In such a case, the Assembler prints $END ASMB PASS on the console device at the end of pass 1. The operator responds by reloading the source input into the non-disc device and then entering GO, ASMB through the console device.

1-2. SYMBOLIC ADDRESSING

Symbols may be used for referring to machine instructions, data, constants, and certain other pseudo operations. A symbol represents the address for a computer word in memory. A symbol is defined when it is used as a label for a location in the program, a name of a common storage segment, the label of a data storage area or constant, the label of an absolute or relocatable value, or a location external to the program.

Through use of simple arithmetic operators, symbols may be combined with other symbols or numbers to form an expression which may identify a location other than that specifically named by a symbol. Symbols appearing in operand expressions, but not specifically defined, and symbols that are defined more than once are considered to be in error by the Assembler.

1-3. MEMORY ADDRESSING

1-4. PAGING

The computer memory is logically divided into pages of 1024 words each. A page is defined as the largest block of memory which can be addressed directly by the memory address bits of a memory reference instruction (single-length). These memory reference instructions have 10 bits to specify a memory address, and thus the page size is 1024 locations (2000 octal). Octal addresses for each page, up to the maximum memory size, are shown in table 1-1.

Provision is made to address directly one of two pages: page zero (the base page, consisting of locations 00000_8 through 01777_8), and the current page (the page in which the instruction itself is located). Memory reference instructions include a bit (bit 10) reserved to specify one or

the other of these two pages. To address locations in any other page, indirect addressing is used. Page references are specified by bit 10 as follows:

Logic 0 = page zero (Z)
Logic 1 = current page (C)

1-5. INDIRECT ADDRESSING

All memory reference instructions reserve a bit to specify direct or indirect addressing. For single-length memory reference instructions, bit 15 of the instruction word is used; for extended arithmetic memory reference instructions, bit 15 of the address word is used. Indirect addressing uses the address part of the instruction to access another word in memory, which is taken as a new memory reference for the same instruction. This new address word is a full 16 bits long, 15 bits of address plus another direct-indirect bit. The 15-bit length of the address permits access to any location in memory. If bit 15 again specifies indirect addressing, still another address is obtained. This multiple-step indirect addressing may be done to any number of levels. The first address obtained in the indirect phase which does not specify another indirect level becomes the effective address for the instruction. Direct or indirect addressing is specified by bit 15 as follows:

Logic 0 = direct
Logic 1 = indirect

1-6. PROGRAM RELOCATION

Relocatable programs are relocated at absolute addresses by the relocating loader.

Relocatable code assumes a starting location of 00000, and this location is termed the relative, or *relocatable* origin. The absolute origin (termed the relocation base) of a relocatable program is determined by the loader. The value of the absolute origin is added to the zero-relative value of each operand address to obtain the absolute operand address. The absolute origin, and thus the values of every operand address, may vary each time the program is loaded.

A relocatable program may be composed of several independently assembled or compiled subprograms. Each of the subprograms will have a relative origin of 00000. Each subprogram is then assigned a unique absolute origin upon being loaded.

The operand values produced by the Assembler may be *program* relocatable, *base page* relocatable, or *common* relocatable. Each of these segments of the program has a separate relocation base or origin. Operands that are references to locations in the main portion of the program are incremented by the program relocation base; those referring to the base page, by the base page relocation base; and those referring to common storage, by the common relocation base.

If the loader or system generator encounters an operand that is a reference to a location in a page other than the current page or base page, a link is established. A link is a word in the base page or current page which is allocated to contain the full 15-bit address of the referenced location. The address of the link is then substituted as an indirect address in the instruction in the current page. If other similar references are made to the same location, they are linked through the same link.

Table 1-1. Logical Memory Address/Pages

MEMORY SIZE	PAGE	OCTAL ADDRESSES
4K	0	00000 to 01777
	1	02000 to 03777
	2	04000 to 05777
	3	06000 to 07777
8K	4	10000 to 11777
	5	12000 to 13777
	6	14000 to 15777
	7	16000 to 17777
12K	8	20000 to 21777
	9	22000 to 23777
	10	24000 to 25777
	11	26000 to 27777
16K	12	30000 to 31777
	13	32000 to 33777
	14	34000 to 35777
	15	36000 to 37777
24K	16	40000 to 41777
	17	42000 to 43777
	18	44000 to 45777
	19	46000 to 47777
	20	50000 to 51777
	21	52000 to 53777
	22	54000 to 55777
	23	56000 to 57777
32K	24	60000 to 61777
	25	62000 to 63777
	26	64000 to 65777
	27	66000 to 67777
	28	70000 to 71777
	29	72000 to 73777
	30	74000 to 75777
	31	76000 to 77777

1-7. PROGRAM LOCATION COUNTER

The Assembler maintains a counter, called the program location counter, that assigns consecutive memory addresses to source statements.

The initial value of the program location counter is established according to the use of either the NAM or ORG pseudo operation at the start of the program. The NAM operation causes the program location counter to be set to zero for a relocatable program; the ORG operation specifies the absolute starting location for an absolute program.

Through use of the ORB pseudo operation a relocatable program may specify that certain operations or data areas be allocated to the base page. If so, a separate counter, called the base page location counter, is used in assigning these locations.

1-8. SOURCE PROGRAM

Figure 1-1 shows an assembler coding form and the code for a simple program which counts the number of 1's and 0's in the A-register. The first statement is the control statement, and contains the assembly options R (for a relocatable source program), B (the program is to be punched on the standard punch device in binary form), L (a program listing is to be printed on the standard list device), and T (a listing of the symbol table is to be printed on the standard list device). See paragraph 1-9 and table 1-2 for a further discussion of control statement parameters.

Following the control statement, the first statement of the program (other than remarks or a HED statement) must be a NAM statement for a relocatable program or an ORG statement to indicate the origin of an absolute program. The last statement must be an END statement and may contain a transfer address for the start of a relocatable program. Each statement is terminated by an end-of-statement or end-of-record mark if not on cards.

1-9. ASSEMBLY OPTIONS

The control statement must be the first statement in the source program and it specifies the desired assembly options:

$$\text{ASMB},p_1,p_2,\ldots,p_n$$

"ASMB," is in positions 1-5 of the statement. Following the comma are one or more parameters, in any order: The parameters are shown in table 1-2. If output to the LG Area is specified in the ON,ASMB directive (ON,ASMB, . . . ,99), the control statement does not require the B parameter for relocatable output to be generated onto the LG Area. (Only specify B if punched form required.)

Since they contradict one another, F and X must never appear in the control statement for the same source program. If neither A nor R is specified, R is assumed. If T is omitted, the symbol table listing will *not* be printed. If B is omitted, the object program will *not* be punched on the standard punch output device (it may, however, be retained in the LG Area of the disc if so specified in the ON,ASMB RTE directive).

"ASMB" alone or with either A or R as the only option specified, will direct the Assembler to process the source information without producing any output. Error messages will be printed on the list device, however. Thus, the user may use this method to examine the source for errors prior to producing the final object code.

1-10. BINARY OUTPUT

The binary output is defined by the ASMB control statement. The binary output includes the object code for the instructions translated from the source program. It does not include system subroutines referenced within the source program (arithmetic subroutine calls, .IOC., .DIO., .ENTR, etc.)

1-11. SYMBOL TABLE

Figure 1-2 shows the symbol table listing produced when the source program was assembled. Columns 1 through 5 contain the name of the label. The R in column 7 specifies that the source program is relocatable and columns 9 through 14 contain the location (in octal) where the label symbol is stored. (In the example shown in figure 1-2, the locations are relative because the source program is relocatable.)

The characters that designate an external symbol or type of relocation for the Operand field or the symbol are as follows:

Character	Relocation Base
Blank	Absolute
R	Program relocatable
C	Common relocatable
X	External symbol
B	Base page relocatable
S	Substitution code

HEWLETT-PACKARD ASSEMBLER CODING FORM

PROGRAMMER | DATE | PROGRAM | PAGE OF

STATEMENT

Label	Operation	Operand	Comment
ASMB, R, L, B			
	NAM	COUNT	
	ENT	COUNT	
* A PROGRAM TO COUNT THE NUMBER OF ONES IN THE A-REGISTER			
COUNT	NOP		
	CLB		CLEAR B (B USED AS COUNT OF 1'S)
	LDX	=D16	LOAD 16 INTO X (WITH LITERAL)
LOOP	SLA		IS A-REGISTER BIT Ø ONE?
	INB		YES, ADD 1 TO COUNT IN B
	RAL		ROTATE A-REGISTER LEFT 1
	DSX		DECREMENT X & SKIP IF Ø (DONE?)
	JMP	LOOP	NOT DONE, REPEAT
	JMP	COUNT, I	RETURN COUNT IS IN B-REGISTER
	END		

Ø = ZERO | O = ALPHA O | 1 OR I = ONE | 2 = TWO | I = ALPHA I | Z = ALPHA Z | LINE TERMINATED BY RETURN, LINE FEED (R LF) | LINE IS DELETED BY RUBOUT BEFORE R LF

M0255 5080-6596

Figure 1-1. Source Program

Table 1-2. Control Statement Parameters

PARAMETER	MEANING
A	Absolute assembly. The addresses generated by the Assembler are to be interpreted as absolute locations in memory. The program is a complete entity; external symbols, common storage references and entry points are not permitted. Note that an absolute program *cannot* be executed on RTE.
R	Relocatable assembly. The object program may be loaded anywhere in memory. All operands which refer to memory locations are automatically adjusted as the program is loaded. Operands referring to memory locations greater than 1777_8 must be relocatable expressions. Programs may contain external symbols and entry points, and may refer to common storage.
B	Binary output. An absolute or relocatable object program is to be output on the standard punch device.
L	List output. A program listing is to be printed on the standard list device. Error messages will still be printed if "L" is not specified.
T	Symbol table print. A listing of the symbol table is to be printed on the standard list output device.
N,Z	Selective assembly. Sections of the program are to be included or excluded at assembly time depending upon the option specified. See the descriptions of the IFN and IFZ pseudo instructions in Section IV of this manual.
C	Cross reference table print. All references to statement labels, external symbols, and user-defined opcodes are to be listed on the standard list output device after the end of the assembly.
F	Floating point instructions. The floating point machine instructions are to be used instead of the software simulation routines for the following floating point operations: FIX, FLT, FDV, FMP, FAD, and FSB.
X	No EAU hardware. Signifies that the object program will be executed on a machine which does *not* have the Extended Arithmetic Unit (EAU) hardware. This parameter prevents the use of the following EAU instructions: ASR, ASL, RRR, RRL, LSR, LSL, and SWP. In addition, it causes all occurrences of the MPY, DIV, DLD, and DST instructions to be substituted with a call to the appropriate subroutine in the relocatable library.

```
PAGE 0001


0001                          ASMB,R,B,L,T
LOOP   R 000001
COUNT  R 000005
BIT0   R 000010
BIT1   R 000013
BIT2   R 000016
MORE   R 000022
BIT3   R 000023
LESS1  R 000024
LESS2  R 000026
EVEN   R 000027
**  NO ERRORS*
```

Figure 1-2. Symbol Table Listing

1-12. LIST OUTPUT

Columns	Content
1-4	Source statement sequence number generated by the Assembler
5-6	Blank
7-11	Location (octal)
12	Blank
13-18	Object code word in octal
19	Relocation or external symbol indicator
20	Blank
21-80	First 60 characters of source statement

Lines consisting entirely of comments (i.e., * in column 1) are printed as follows:

Columns	Content
1-4	Source statement sequence number
5-80	Up to 76 characters of comments

At the end of each pass, the following is printed on the list device:

```
Pass 1 =
** NO ERRORS PASS#1 **RTE ASMB xxxxx-yyyyy**

                    or

**nnnn ERRORS PASS#1 **RTE ASMB xxxxx-yyyyy**

Pass 2 =
** NO ERRORS *TOTAL **RTE ASMB xxxxx-yyyyy**

                    or

**nnnn ERRORS *TOTAL **RTE ASMB xxxxx-yyyyy**
```

The value *nnnn* indicates the number of errors. Pass 2 error count includes the total error count of pass 1 and pass 2. xxxxx-yyyyy is the Assembler's part number.

If there are errors, the message PG *xxx* is printed on the list device immediately preceding the **nnnn ERRORS* message, where *xxx* is the page number where the final error was detected. The same message appears in the listing following each error and it points to the page number where the previous error was detected. The backwards pointer following the first error in the program is PG 000.

SOURCE STATEMENT FORMAT

SECTION II

A source language statement consists of a label, an operation code, an operand (or operands) and comments. The label is used when needed as a reference by other statements. The operation code may be a mnemonic machine operation or an assembly directing pseudo code. An operand may be an expression consisting of an alphanumeric symbol, a number, a special character, or any of these combined by arithmetic operators. An operand may also be a literal. Indicators may be appended to an operand to specify certain functions such as indirect addressing. The comments portion of the statement is optional.

2-1. STATEMENT OF CHARACTERISTICS

The fields of the source statement appear in the following order:

1. Label
2. Opcode
3. Operands
4. Comments

2-2. FIELD DELIMITERS

One or more spaces separate the fields of a statement. A single space as the first character of a statement signifies that there is no label for this statement.

2-3. CHARACTER SET

The characters that may appear in a statement are as follows:

A through Z
0 through 9
. (period)
* (asterisk)
+ (plus)
- (minus)
, (comma)
= (equals)
() (parentheses)
(space)

Any other ASCII characters may appear in the Comments field. (See Appendix A.)

The letters A through Z, the numbers 0 through 9, and the period may be used in an alphanumeric symbol. In the first position in the Label field, an asterisk indicates a comment; in the Operand field, it represents the value of the program location counter for the current instruction. The plus and minus are used as operators in arithmetic address expressions. The comma separates several operation codes, or an expression and an indicator in the Operand field. An equals sign indicates a literal value. The parentheses are used only in the COM pseudo instruction.

Spaces separate fields of a statement and operands in a multi-operand field. They may also be used to enhance the appearance of the listing. Within a field they may be used freely when following +, −, ,, or (.

2-4. STATEMENT LENGTH

A statement may contain up to 80 characters including blanks, but excluding the end-of-statement mark.

2-5. LABEL FIELD

The Label field identifies the statement and may be used as a reference point by other statements in the program.

The field starts in position one of the statement. It is terminated by space. A space in position one signifies that the statement is unlabeled.

2-6. LABEL SYMBOL

A label may have one to five characters consisting of A through Z, 0 through 9, and the period.

Note: The Assembler allows the use of certain other characters in the Label field. However, they are reserved for use in Hewlett-Packard programs.

The first character must be alphabetic or a period. A label of more than five characters could be entered on the source statement, but the Assembler flags this condition as an error and truncates the label from the right to five characters. Some examples are shown in figure 2-1.

Each label must be unique within the program; two or more statements may not have the same symbolic name. Names which appear in the Operand field of an EXT or COM pseudo instruction may not also be used as statement labels

in the same subprogram. However, names appearing in a COM pseudo instruction may be defined as entry points in an ENT pseudo instruction. Some examples are shown in figure 2-2.

2-7. ASTERISK

An asterisk in position one indicates that the entire statement is a comment. Positions 2 through 80 are available; however, positions 1 through 76 only are printed as part of the assembly listing. An asterisk within a label is illegal in any position

2-8 OPCODE FIELD

The operation code defines an operation to be performed by the computer or the Assembler. The Opcode field follows the Label field and is separated from it by at least one space. If there is no label, the operation code may begin anywhere after position one. The Opcode field is terminated by a space immediately following an operation code. Operation codes are organized in the following categories:

Machine operation codes:

- Memory Reference
- Register Reference
- Input/Output, Overflow, and Halt
- Extended Arithmetic Unit
- Floating Point
- Memory Mapping
- Decimal Arithmetic

Pseudo operation codes:

- Assembler control
- Object program linkage
- Address and symbol definition
- Constant definition
- Storage allocation
- Arithmetic subroutine calls
- Assembly Listing Control
- Define User Opcodes
- Code-replacement definition

Operation codes are discussed in detail in Sections III and IV.

2-9. OPERAND FIELD

The meaning and format of the Operand field depend on the type of operation code used in the source statement. The field follows the Opcode field and is separated from it by at least one space. If more than one operand is required, they are separated from one another by at least one space.

An Operand may contain an expression consisting of one of the following:

- Single symbolic term
- Single numeric term
- Asterisk
- Combination of symbolic terms, numeric terms, and the asterisk joined by the arithmetic operators + and −.

An expression may be followed by a comma, an indirect addressing indicator (see paragraph 2-20), and a Clear Flag indicator (see paragraph 2-21). Programs may also contain a literal value in the Operand field. (See paragraph 2-19.)

2-10. SYMBOLIC TERMS

A symbolic term may be one to five characters consisting of A through Z, 0 through 9, and the period. The first character must be alphabetic or a period. Some examples are shown in figure 2-3.

A symbol used in the Operand field must be a symbol that is defined elsewhere in the program in one of the following ways:

- As a label in the Label field of a machine operation or a user-defined instruction
- As a label in the Label field of a BSS, ASC, DEC, DEX, OCT, DEF, BYT, ABS, EQU, DBL, DBR or REP pseudo operation
- As a name in the Operand field of a COM or EXT pseudo operation
- As a label in the Label field of an arithmetic subroutine pseudo operation

The value of a symbol is absolute or relocatable depending on the assembly option selected by the user. The Assembler assigns a value to a symbol as it appears in one of the above fields of a statement. If a program is to be loaded in absolute form, the values assigned by the Assembler remain fixed. If the program is to be relocated, the actual value of a symbol is established on loading. A symbol may be assigned an absolute value through use of the EQU pseudo instruction.

A symbolic term may be preceded by a plus or minus sign. If preceded by a plus or no sign, the symbol refers to its associated value. If preceded by a minus sign, the symbol refers to the two's complement of its associated value. A single negative symbolic operand may be used only with the ABS pseudo operation.

HEWLETT-PACKARD ASSEMBLER CODING FORM

PROGRAMMER | DATE | PROGRAM

STATEMENT

Label	Operation	Operand	Comments
	LDA		NO LABEL
.ABCD			VALID LABEL
.1234			VALID LABEL
A.123			VALID LABEL
.			VALID LABEL
1.ABC			ILLEGAL LABEL - FIRST CHARACTER NUMERIC
ABC123			ILLEGAL LABEL - TOO LONG TRUNCATED TO ABC12
A*BC			ILLEGAL LABEL - ASTERISK NOT ALLOWED IN LABEL
ABC			NO LABEL - SPACE IN FIRST POSITION - ASSEMBLER ATTEMPTS TO INTERPRET ABC AS AN OPCODE

Computer Museum

Figure 2-1. Label Examples

HEWLETT-PACKARD ASSEMBLER CODING FORM

PROGRAMMER | DATE | PROGRAM

STATEMENT

Label	Operation	Operand	Comments
	COM	ACOM(20), BC(30)	
LB	EQU	160	VALID LABEL
	ENT	ABC	
	EXT	XL1, XL2	
START	LDA	LB	VALID LABEL
N25			VALID LABEL
XL2			ILLEGAL LABEL - USED IN EXT
BC			ILLEGAL LABEL - USED IN COM
N25			ILLEGAL LABEL - PREVIOUSLY DEFINED

Figure 2-2. Label Usage Examples

2-11. NUMERIC TERMS

A numeric term may be decimal or octal. A decimal number is represented by one to five digits within the range 0 to 32767. An octal number is represented by one to six octal digits followed by the letter B (0 to 177777B).

If a numeric term is preceded by a plus or no sign, the binary equivalent of the number is used in the object code. If preceded by a minus sign, the two's complement of the binary equivalent is used. A negative numeric operand may be used only with the DEX, DEC, OCT, BYT and ABS pseudo operations.

For a memory reference instruction in an absolute program, the maximum value of a numeric operand depends on the type of machine or pseudo instruction. In a relocatable program, the value of a numeric operand may not exceed 1777_8. Numeric operands are absolute. Their value is not altered by the assembler or the loader.

2-12. ASTERISK

An asterisk in the Operand field refers to the value in the program location counter at the time the source program statement is encountered. The asterisk is considered a relocatable term in a relocatable program.

2-13. EXPRESSION OPERATORS

The asterisk, symbols, and numbers may be joined by the arithmetic operators + and − to form arithmetic address expressions. The Assembler evaluates an expression and produces an absolute or relocatable value in the object code. Some examples are shown in figure 2-4.

2-14. EVALUATION OF EXPRESSIONS

An expression consisting of more than one operand is reduced to a single value. In expressions containing more than one operator, evaluation of the expression proceeds from left to right. The algebraic expression A-(B−C+5) must be represented in the Operand field as A−B+C−5. Parentheses are not permitted in operand expressions.

The range of values that may result from an operand expression depends on the type of operation. The Assembler evaluates expressions as follows:†

Pseudo Operations:	modulo 2^{15}-1
Memory Reference:	modulo 2^{10}-1
Input/Output:	2^6 - 1 (maximum value)

2-15. EXPRESSION TERMS

The terms of an expression are the numbers and the symbols appearing in it. Decimal and octal integers, and symbols defined as being absolute in an EQU pseudo operation are absolute terms. The asterisk and all symbols that are defined in the program are relocatable or absolute depending on the type of assembly. (RTE Assembler allows externals with offset and indirect external references.)

Within a relocatable program, terms may be program relocatable or common relocatable or base page relocatable. A symbol that names an area of common storage is a common relocatable term. A symbol that is defined in any statement other than COM or EQU is a relocatable term. Within one expression all relocatable terms must be program relocatable, common relocatable or base page relocatable; the types may not be mixed.

2-16. ABSOLUTE AND RELOCATABLE EXPRESSIONS

An expression is absolute if its value is unaffected by program relocation. An expression is relocatable if its value changes according to the location into which the program is loaded. In an absolute program, all expressions are absolute. In a relocatable program, an expression may be program relocatable, common relocatable, base page relocatable, or absolute (if less than 2000_8) depending on the definition of the terms composing it.

2-17. ABSOLUTE EXPRESSIONS. An absolute expression may be any arithmetic combination of absolute terms. It may contain relocatable terms alone, or in combination with absolute terms. If relocatable terms appear, there must be an even number of them; they must be of the same type; and they must be paired by sign (a negative term for each positive term). The paired terms do not have to be contiguous in the expression. The pairing of terms by type cancels the effect of relocation; the value represented by a pair remains constant.

An absolute expression reduces to a single absolute value. The value of an absolute multi-term expression may be negative only for ABS pseudo operations. A single numeric term also may be negative in an OCT, DEX, BYT, or DEC pseudo instruction. In a relocatable program the value of an absolute expression must be less than 2000_8 for instructions that reference memory locations (Memory Reference, DEF, Arithmetic subroutine calls, etc.).

If P_1 and P_2 are program relocatable terms; C_1 and C_2, common relocatable; and A, an absolute term; then the following are absolute terms:

$A - C_1 + C_2$	$A - P_1 + P_2$	$C_1 - C_2 + A$
$A + A$	$P_1 - P_2$	$-C_1 + C_2 + A$
$* - P_1$	$-P_1 + P_2$	$-A - P_1 + P_2$

†The evaluation of expressions by the Assembler is compatible with the addressing capability of the hardware instructions (e.g., up to 32K words through Indirect Addressing). The user must take care not to create addresses which exceed the memory size of the particular configuration.

HEWLETT-PACKARD ASSEMBLER CODING FORM

PROGRAMMER | DATE | PROGRAM

STATEMENT

Label	Operation	Operand	Comments
	LDA	A1234	VALID OPERAND
	ADA	B.1	VALID OPERAND
	JMP	ENTRY	VALID OPERAND
	LDA	A1234+B.1-ENTRY	VALID OPERAND
	STA	1ABC	ILLEGAL OPERAND - FIRST CHARACTER NUMERIC
	STA	ABCDEF	ILLEGAL OPERAND - MORE THAN FIVE CHARACTERS

Figure 2-3. Symbolic Operand Examples

```
LDA SYM+6        ADD 6 TO THE VALUE OF SYM
ADA SYM-3        SUBTRACT 3 FROM THE VALUE OF SYM
 .
 .
 .
JMP *+5          ADD 5 TO THE CONTENTS OF THE
 .               PROGRAM LOCATION COUNTER.
 .
 .
STB -A+C-4       ADD - VALUE OF A, THE VALUE OF C
 .               AND SUBTRACT 4.
 .
 .
STA XTA-*        SUBTRACT VALUE OF PROGRAM
                 LOCATION COUNTER FROM VALUE OF
                 XTA.
```

Figure 2-4. Expression Operator Examples

The asterisk is program relocatable.

2-18. RELOCATABLE EXPRESSIONS. A relocatable expression is one whose value is changed by the loader. All relocatable expressions must have a positive value.

A relocatable expression may contain an odd number of relocatable terms, alone, or in combination with absolute terms. All relocatable terms must be of the same type. Terms must be paired by sign with the odd term being positive.

A relocatable expression reduces to a single positive relocatable term, adjusted by the values represented by the absolute terms and paired relocatable terms associated with it.

If P_1, P_2, and P_3 are program relocatable terms; C_1, C_2 and C_3, common relocatable; and A, an absolute term; then the following are relocatable terms:

$P_1 - A$	$C_1 - A$	$P_1 - P_2 + *$
$P_1 - P_2 + P_3$	$C_1 - C_2 + C_3$	$C_1 + A$
$* + A$	$* - P_1 + P_2$	$* - A$
$P_2 + A$	$A + C_1$	$- A - P_1 + P_2 + P_3$
$P_1 - *$	$C_1 - C_2 + C_3 - A$	$A + *$
		$- C_1 + C_2 + C_3$

2-19. LITERALS

Literal values may be specified as operands in relocatable programs. (Literals are not allowed in absolute programs.) The Assembler converts the literal to its binary value, assigns an address to it, and substitutes this address as the operand. Locations assigned to literals are those immediately following the last location used by the program.

A literal is specified by using an equal sign and a one-character identifier defining the type of literal. The actual literal value is specified immediately following this identifier; no spaces may intervene.

The identifiers are:

=D a decimal integer, in the range -32767 to 32767, including zero.†

=F a floating point number; any positive or negative real number in the range 10^{-38} to 10^{38}, including zero.†

=B an octal integer, one to six digits, $b_1 b_2 b_3 b_4 b_5 b_6$, where b_1 may be 0 or 1, and b_2-b_6 may be 0 to 7.†

=A two ASCII characters.†

=L an expression which, when evaluated, will result in an absolute value. All symbols appearing in the expression must be previously defined.

If the same literal is used in more than one instruction or if different literals have the same value (e.g., =B100 and =D64), only one value is generated, and all instructions using these literals refer to the same location.

Literals may be specified only in the following memory reference, register reference, EAU, and pseudo instructions:

ADA	CPA	MBT	may use =D, =B, =A, =L
ADB	CPB	JRS	
ADX	DIV	MPY	
ADY	IOR	MVW	
AND	LDA	SBS	
CBS	LDB	TBS	
CBT	LDX	XOR	
CMW	LDY		
DLD	FDV	FSB	may use =F
FMP	FAD		

Examples are as follows:

LDA	=D7980	A-Register is loaded with the binary equivalent of 7980_{10}.
IOR	=B777	Inclusive OR is performed with contents of A-Register and 777_8.
LDA	=ANO	A-Register is loaded with binary representation of ASCII characters NO.
LDB	=LZETZ-ZOOM+68	B-Register is loaded with the absolute value resulting from the expression.
FMP	=F39.75	Contents of A- and B-Registers multiplied by floating point constant 39.75.

2-20. INDIRECT ADDRESSING

The HP computers provide an indirect addressing capability for memory reference instructions. The operand portion of an indirect instruction contains the address of another location. The secondary location may be the operand or it may be indirect also and give yet another location, and so forth. The chaining ceases when a location is encountered that does not contain an indirect address. Indirect addressing provides a simplified method of address modifications as well as allowing access to any location in core. See Section I, paragraph 1-5 for a further discussion of indirect addressing.

The Assembler allows specification of indirect addressing by appending a comma and the letter I to any memory

†See CONSTANT DEFINITION, Section IV.

reference operand. The actual address of the instruction may be given in a DEF pseudo operation; this pseudo operation may also be used to indicate further levels of indirect addressing. An example is shown in figure 2-5.

A relocatable assembly language program, however, may be designed without concern for the pages in which it will be stored; indirect addressing is not required in the source language. When the program is being loaded, the loader provides indirect addressing whenever it detects an operand which does not fall in the current page or the base page. The loader substitutes a reference to a program link location (established by the loader in either the base page or the current page) and then stores an indirect address in the particular program link location. If the program link location is in the base page, all references to the same operand from other pages will be via the same link location.

Note: The Basic Control System provides program links to the base page only (not the current page).

2-21. CLEAR FLAG INDICATOR

The majority of the input/output instructions can alter the status of the input/output interrupt flag after execution or after the particular test is performed. In source language, this function is selected by appending a comma and a letter C to the Operand field. Some examples are shown in figure 2-6.

2-22. COMMENTS FIELD

The Comments field allows the user to transcribe notes on the program that will be listed with source language coding on the output produced by the Assembler. The field follows the Operand field and is separated from it by at least one space. The end-of-record mark, the end-of-statement mark, (CR) (LF) , or the 80th character of a statement terminates the field. The statement length should not exceed 60 characters, the width of the source language portion of the listing. A whole line (up to 76 characters), however, can be specified as a comment by inserting an asterisk in the first position. On the list output, statements consisting entirely of comments begin in position 5 rather than 21 as with other source statements. Any characters beyond the above limits will not appear on the listing.

If there is no operand present, the Comments field should be omitted in the NAM and END pseudo operations and in the input/output statements, SOC, SOS, and HLT. If a comment is used, the Assembler attempts to interpret it as an operand. This limitation applies also to multi-operand instructions.

```
AB      LDA SAM,I      EACH TIME THE ISZ IS EXECUTED,
AC      ADA SAM,I      THE EFFECTIVE OPERAND OF AB AND
AD      ISZ SAM        AC CHANGE ACCORDINGLY.
         .
         .
         .
SAM     DEF ROGER
```

Figure 2-5. Indirect Addressing Example

```
STC   13B,C   SET CONTROL AND CLEAR THE FLAG OF SELECT CODE 13 (OCTAL)
OTB   16B,C   CLEAR FLAG OF SELECT CODE 16 (OCTAL) ALONG WITH OUTPUT TO DEVICE
```

Figure 2-6. Clear Flag Examples

MACHINE INSTRUCTIONS

SECTION III

The Assembler language machine instruction codes take the form of three-letter mnemonics. Each source statement corresponds to a machine operation in the object program produced by the Assembler.

Notation used in representing source language instruction is as follows:

label	Optional statement label
m	Memory location — an expression
I	Indirect addressing indicator
sc	Select code — an expression
C	Clear interrupt flag indicator
comments	Optional comments
[]	Brackets defining a field or portion of a field that is optional
{ }	Brackets indicating that one of the set may be selected.
lit	literal

3-1. MEMORY REFERENCE

The memory reference instructions perform arithmetic, logical, jump, word manipulation, byte manipulation, and bit manipulation operations on the contents of memory locations and the registers. An instruction may directly address the 2048_{10} words of the current and base pages. If required, indirect addressing may be used to refer to all $32{,}768_{10}$ words of memory. Expressions in the Operand field are evaluated modulo 2^{10}.

External memory references may be made with + or − offsets, with indirects or both.

If the program is to be assembled in relocatable form, the Operand field may contain relocatable or absolute expressions; however, absolute expressions must be less than 2000_8 in value. If the program is to be assembled in absolute form, the Operand field may contain any expression which is consistent with the location of the program. Literals may not be used in absolute programs. Absolute programs must be complete entities; they may not refer to external subroutines or to common storage.

3-2. JUMP AND INCREMENT-SKIP

Jump and Increment-Skip instructions may alter the normal sequence of program execution.

label	JMP	m [,I]	comments

Jump to m. Jump indirect inhibits interrupt until the transfer of control is complete, or three levels of indirecting have occurred.

label	JSB	m [,I]	comments

Jump to subroutine. The address for label + 1 is placed into the location represented by m and control transfers to m + 1. On completion of the subroutine, control may be returned to the normal sequence by performing a JMP m,I.

label	ISZ	m [,I]	comments

Increment, then skip if zero. ISZ adds 1 to the contents of m. If m then equals zero, the next instruction in memory is skipped.

3-3. ADD, LOAD AND STORE

Add, Load, and Store instructions transmit and alter the contents of memory and of the A- and B-Registers. A litera, indicated by "lit", may be either =D, =B, =A, or =L type. See Section II, paragraph 2-19 for a further discussion of literals.

label	ADA	{ m [,I] / lit }	comments

Add the contents of m to A.

label	ADB	{ m [,I] / lit }	comments

Add the contents of m to B.

label	LDA	{ m [,I] / lit }	comments

Load A with the contents of m.

label	LDB	{ m [,I] / lit }	comments

Load B with the contents of m.

label	STA	m [,I]	comments

Store contents of A in m.

label	STB	m [,I]	comments

Store contents of B in m.

In each instruction, the contents of the sending location is unchanged after execution.

3-4. LOGICAL OPERATIONS

The logical instructions allow bit manipulation and the comparison of two computer words.

label	AND	{ m [,I] / lit }	comments

The logical product ("AND") of the contents of m and the contents of A are placed in A.

label	XOR	{ m [,I] / lit }	comments

The modulo-two sum (exclusive "or") of the bits in m and the bits in A is placed in A.

label	IOR	{ m [,I] / lit }	comments

The logical sum (inclusive "or") of the bits in m and the bits in A is placed in A.

label	CPA	{ m [,I] / lit }	comments

Compare the contents of m with the contents of A. If they differ, skip the next instruction; otherwise, continue.

label	CPB	{ m [,I] / lit }	comments

Compare the contents of m with the contents of B. If they differ, skip the next instruction; otherwise, continue.

3-5. WORD PROCESSING (21MX ONLY)

The word processing instructions allow the user to move a series of data words from one array in memory to another or to compare (word-by-word) the contents of two arrays in memory.

label	MVW	{ literal / m [,I] }	comments

Move words. The A-register contains the starting (lowest) word address of the source array. The B-register contains the starting (lowest) word address of the destination array. These addresses *must not be indirect.* The number of words to be moved is specified by *literal* or by the value contained in *m* [,I]. The specified number of words are moved from the source array into the destination array. As each word is moved, the A- and B-registers are incremented by one. The source array is not altered.

label	CMW	{ literal / m [,I] }	comments

Compare words. The A-register contains the starting (lowest) word address of array #1. The B-register contains the starting (lowest) word address of array #2. These addresses *must not be indirect.* The number of word comparisons to be performed is specified by *literal* or by the value contained in *m* [,I]. The two arrays are compared word-by-word beginning at the specified addresses. The operation is finished when an inequality is detected or when the specified number of word comparisons have been performed. When the operation is finished, the A-register contains the word address of the last word in array #1 which was compared, except when the two arrays are equal. In this case the A-register contains the starting address of array #1, incremented by the count parameter. The B-register contains the starting address of array #2 incremented by the "count" parameter (*literal* or the value in *m* [,I]). If the two arrays are equal, execution proceeds at the next sequential source language instruction (P+3). If array #1 is "less than " #2, execution proceeds at instruction P+4. If array #1 is "greater than" array #2, execution proceeds at instruction P+5. The two arrays are not altered.

3-6. BYTE PROCESSING (21MX ONLY)

The byte processing instructions allow the user to copy a data byte from memory into the A- or B-register, copy a data byte from the A- or B-register into memory, copy a series of data bytes from one array in memory to another, compare (byte-by-byte) the contents of two arrays in memory, or scan an array in memory for particular data bytes.

A *byte address* is defined as two times the word address of the memory location containing the particular data byte. If the byte location is the low order half of the memory location (bits 0-7), bit 0 of the byte address is set; if the byte location is the high order half of the memory location (bits 8-15), bit 0 of the byte address is clear. Byte addresses *may not be indirect.*

label	LBT	comments

Load byte. The B-register contains the byte address of the byte to be loaded. The specified byte is copied from memory into bits 0-7 of the A-register (bits 8-15 of the A-register are set to zeros). The B-register is then incremented by one. The memory location is not altered.

label	SBT	comments

Store byte. The B-register contains the byte address into which the byte is to be stored. Bits 0-7 of the A-register are copied into the specified memory byte location (bits 8-15 of the A-register are ignored). The B-register is then incremented by one. The A-register is not altered.

label	MBT	{ literal / m [,I] }	comments

Move bytes. The A-register contains the starting (lowest) byte address of the source array. The B-register contains the starting (lowest) byte address of the destination array. The number of bytes to be moved is specified by *literal* or by the value contained in *m* [,I]. The specified number of bytes are moved from the source array into the destination array. As each byte is moved, the A- and B-registers are incremented by one. The source array is not altered.

label	CBT	{ literal / m [,I] }	comments

Compare bytes. The A-register contains the starting (lowest) byte address of array #1. The B-register contains the starting (lowest) byte address of array #2. The number of byte comparisons to be performed is specified by *literal* or by the value contained in *m* [,I]. The two arrays are compared byte-by-byte beginning at the specified addresses. The operation is finished when an inequality is detected or when the specified number of byte comparisons have been performed. When the operation is finished, the A-register contains the byte address of the last byte in array #1 where the comparison stopped; the B-register contains the starting byte address of array #2 incremented by the "count" parameter (*literal* or the value in *m* [,I]). If the two arrays are equal, execution proceeds at the next sequential source language instruction (P+3). If array #1 is "less than" array #2, execution proceeds at instruction P+4. If array #1 is "greater than" array #2, execution proceeds at instruction P+5. The two arrays are not altered.

label	SFB	comments

Scan for byte. The A-register contains a test byte in bits 0-7 and a termination byte in bits 8-15. The B-register contains the starting (lowest) byte address of the array to be scanned. The array is compared byte-by-byte against both the test and termination bytes starting at the specified address. The operation is finished when a positive comparison is detected or when the end of memory is reached. If the test byte is detected, execution proceeds at the next sequential source language instruction (P+1) and the B-register contains the address of the test byte in the array. If the termination byte is detected, execution proceeds at instruction P+2 and the B-register contains the address plus one of the termination byte in the array.

The scanning operation will not continue indefinitely even if neither the termination byte nor test byte exists in memory. These bytes are in the A-register with byte addresses 000 and 001, respectively. Thus, if no match is made by the time the B-register points to the last byte in memory, the B-register will roll over to zero and the next test will match the termination byte in the A-register with itself.

3-7. BIT PROCESSING (21MX ONLY)

The bit processing instructions allow the user to selectively test, set, or clear bits in a memory location according to the contents of a mask. In the descriptions below, *addr1* and *addr2* may be operand expressions.

label	TBS	{ literal / addr1[,I] }	addr2[,I]	comments

Test bits. *literal* is a test mask, *addr1* [,I] is the address of a memory location containing a test mask, and *addr2* [,I] is the address of a memory location containing the bits to be tested. The bits in *addr2* [,I] which correspond to the "1" bits in the mask are tested. All other bits in *addr2* [,I] are ignored. If all the tested bits in *addr2* [,I] are set, execution proceeds at the next sequential source language instruction (P+3). If any of the tested bits in *addr2* [,I] are clear, execution proceeds at instruction P+4.

label	SBS	{ literal / addr1[,I] }	addr2[,I]	comments

Set bits. *literal* is a mask, *addr1* [,I] is the address of a memory location containing a mask, and *addr2* [,I] is the address of a memory location containing the bits to be set. The bits in *addr2* [,I] which correspond to the "1" bits in the mask are set. All other bits in *addr2* [,I] are not affected. Functionally, the SBS instruction is a "logical OR" operation.

label	CBS	{ literal / addr1[,I] }	addr2[,I]	comments

Clear bits. *literal* is a mask, *addr1*[,I] is the address of a memory location containing a mask, and *addr2*[,I] is the address of a memory location containing the bits to be cleared. The bits in *addr2*[,I] which correspond to the "1" bits in the mask are cleared. All other bits in *addr2*[,I] are not affected.

3-8. REGISTER REFERENCE

The register reference instructions include a shift-rotate group, an alter-skip group, an index register group, and NOP (no operation). For the shift-rotate and alter-skip groups, the instruction mnemonics within each group may be combined into a single source statement to cause multiple operations to be executed during one memory cycle. In such cases, successive mnemonics within a single source statement are separated from one another by a comma.

3-9. SHIFT-ROTATE GROUP

This group contains 19 basic instructions that can be combined to produce more than 500 different single cycle operations.

CLE	Clear E to zero
ALS	Shift A left one bit, zero to least significant bit. Sign unaltered
BLS	Shift B left one bit, zero to least significant bit. Sign unaltered
ARS	Shift A right one bit, extend sign; sign unaltered
BRS	Shift B right one bit, extend sign; sign unaltered
RAL	Rotate A left one bit
RBL	Rotate B left one bit
RAR	Rotate A right one bit
RBR	Rotate B right one bit
ALR	Shift A left one bit, clear sign, zero to least significant bit
BLR	Shift B left one bit, clear sign, zero to least significant bit
ERA	Rotate E and A right one bit
ERB	Rotate E and B right one bit
ELA	Rotate E and A left one bit
ELB	Rotate E and B left one bit
ALF	Rotate A left four bits
BLF	Rotate B left four bits
SLA	Skip next instruction if least significant bit in A is zero
SLB	Skip next instruction if least significant bit in B is zero

These instructions may be combined as follows:

```
                   ALS                                   ALS
                   ARS                                   ARS
                   RAL                                   RAL
                   RAR                                   RAR
label [            ALR  ] [,CLE] [,SLA] [,            ALR  ]  comments
                   ALF                                   ALF
                   ERA                                   ERA
                   ELA                                   ELA

                   BLS                                   BLS
                   BRS                                   BRS
                   RBL                                   RBL
                   RBR                                   RBR
label [            BLR  ] [,CLE] [,SLB] [,            BLR  ]  comments
                   BLF                                   BLF
                   ERB                                   ERB
                   ELB                                   ELB
```

CLE, SLA, or SLB appearing alone or in any valid combination with each other are assumed to be a shift-rotate machine instruction.

The shift-rotate instructions must be given in the order shown. At least one and up to four are included in one statement. Instructions referring to the A-register may not be combined in the same statement with those referring to the B-register.

3-10. ALTER-SKIP GROUP

The alter-skip group contains 19 basic instructions that can be combined to produce more then 700 different single cycle operations.

CLA	Clear the A-Register
CLB	Clear the B-Register
CMA	Complement the A-Register
CMB	Complement the B-Register
CCA	Clear, then complement the A-Register (set to ones)
CCB	Clear, then complement the B-Register (set to ones)
CLE	Clear the E-Register
CME	Complement the E-Register
CCE	Clear, then complement the E-Register
SEZ	Skip next instruction if E is zero
SSA	Skip if sign of A is positive (0)
SSB	Skip if sign of B is positive (0)

INA	Increment A by one
INB	Increment B by one
SZA	Skip if contents of A equals zero
SZB	Skip if contents of B equals zero
SLA	Skip if least significant bit of A is zero
SLB	Skip if least significant bit of B is zero
RSS	Reverse the sense of the skip instructions. If no skip instructions precede in the statement, skip the next instruction

These instructions may be combined as follows:

label	[{CLA, CMA, CCA}]	[,SEZ]	[,{CLE, CME, CCE}] [,SSA] [,SLA] [,INA] [,SZA] [,RSS]	comments
label	[{CLB, CMB, CCB}]	[,SEZ]	[,{CLE, CME, CCE}] [,SSB] [,SLB] [,INB] [,SZB] [,RSS]	comments

The alter-skip instructions must be given in order shown. At least one and up to eight are included in one statement. Instructions referring to the A-register may not be combined in the same statement with those referring to the B-register. When two or more skip functions are combined in a single operation, a skip occurs if any one of the conditions exists. If a word with RSS also includes both SSA and SLA (or SSB and SLB), a skip occurs only when sign and least significant bit are both set (1).

3-11. INDEX REGISTER GROUP (21MX ONLY)

This group contains 32 instructions which perform various operations involving the use of index registers X and Y. An instruction may directly address the 2048_{10} words of the current and base pages. If required, indirect addressing may be used (except where noted otherwise) to refer to all $32{,}768_{10}$ words of memory. Expressions in the Operand field are evaluated modulo 2^{10}.

label	CAX	comments

Copy A to X. The contents of the A-register are copied into the X-register. The A-register is not altered.

label	CBX	comments

Copy B to X. The contents of the B-register are copied into the X-register. The B-register is not altered.

label	CAY	comments

Copy A to Y. The contents of the A-register are copied into the Y-register. The A-register is not altered.

label	CBY	comments

Copy B to Y. The contents of the B-register are copied into the Y-register. The B-register is not altered.

label	CXA	comments

Copy X to A. The contents of the X-register are copied into the A-register. The X-register is not altered.

label	CXB	comments

Copy X to B. The contents of the X-register are copied into the B-register. The X-register is not altered.

label	CYA	comments

Copy Y to A. The contents of the Y-register are copied into the A-register. The Y-register is not altered.

label	CYB	comments

Copy Y to B. The contents of the Y-register are copied into the B-register. The Y-register is not altered.

label	XAX	comments

Exchange A and X. The contents of the A-register are copied into the X-register and the contents of the X-register are copied into the A-register.

label	XBX	comments

Exchange B and X. The contents of the B-register are copied into the X-register and the contents of the X-register are copied into the B-register.

label	XAY	comments

Exchange A and Y. The contents of the A-register are copied into the Y-register and the contents of the Y-register are copied into the A-register.

label	XBY	comments

Exchange B and Y. The contents of the B-register are copied into the Y-register and the contents of the Y-register are copied into the B-register.

label	ISX	comments

Increment X and skip if zero. The contents of the X-register are incremented by one and then tested. If the new value in X is zero, the next sequential instruction (P+1) is skipped and execution proceeds at instruction P+2; if the new value in X is non-zero, execution proceeds at instruction P+1.

label	ISY	comments

Increment Y and skip if zero. The contents of the Y-register are incremented by one and then tested. If the new value in Y is zero, the next sequential instruction (P+1) is skipped and execution proceeds at instruction P+2; if the new value in Y is non-zero, execution proceeds at instruction P+1.

label	DSX	comments

Decrement X and skip if zero. The contents of the X-register are decremented by one and then tested. If the new value in X is zero, the next sequential instruction (P+1) is skipped and execution proceeds at instruction P+2; if the new value in Y is non-zero, execution proceeds at instruction P+1.

label	DSY	comments

Decrement Y and skip if zero. The contents of the Y-register are decremented by one and then tested. If the new value in Y is zero, the next sequential instruction (P+1) is skipped and execution proceeds at instruction P+2; if the new value in Y is non-zero, execution proceeds at instruction P+1.

label	LDX	{ m [,I] / literal }	comments

Load X from memory. The contents of the specified memory location are copied into the X-register. Indirect addressing may be used. The memory location is not altered.

label	LDY	{ m [,I] / literal }	comments

Load Y from memory. The contents of the specified memory location are copied into the Y-register. Indirect addressing may be used. The memory location is not altered.

label	STX	m [,I]	comments

Store X into memory. The contents of the X-register are copied into the specified memory location. Indirect addressing may be used. The X-register is not altered.

label	STY	m [,I]	comments

Store Y into memory. The contents of the Y-register are copied into the specified memory location. Indirect addressing may be used. The Y-register is not altered.

label	LAX	m [,I]	comments

Load A from memory indexed by X. The contents of the specified memory location are copied into the A-register. Indirect addressing may be used. The address of the memory location is computed by adding the contents of the X-register to *m* or to *m*,I. Note that indirect addressing (if specified) is performed first and then the address is indexed. The X-register and the memory location are not altered.

label	LBX	m [,I]	comments

Load B from memory indexed by X. The contents of the specified memory location are copied into the B-register. Indirect addressing may be used. The address of the memory location is computed by adding the contents of the X-register to *m* or to *m*,I. Note that indirect addressing (if specified) is performed first and then the address is indexed. The X-register and the memory location are not altered.

label	LAY	m [,I]	comments

Load A from memory indexed by Y. The contents of the specified memory location are copied into the A-register. Indirect addressing may be used. The address of the memory location is computed by adding the contents of the Y-register to *m* or to *m*,I. Note that indirect addressing (if specified) is performed first and then the address is indexed. The Y-register and the memory location are not altered.

label	LBY	m [,I]	comments

Load B from memory indexed by Y. The contents of the specified memory location are copied into the B-register. Indirect addressing may be used. The address of the memory location is computed by adding the contents of the Y-register to *m* or to *m*,I. Note that indirect addressing (if specified) is performed first and then the address is indexed. The Y-register and the memory location are not altered.

label	SAX	m [,I]	comments

Store A into memory indexed by X. The contents of the A-register are copied into the specified memory location. Indirect addressing may be used. The address of the memory location is computed by adding the contents of the X-register to *m* or to *m*,I. Note that indirect addressing (if specified) is performed first and then the address is indexed. The A-register and the X-register are not altered.

label	SBX	m [,I]	comments

Store B into memory indexed by X. The contents of the B-register are copied into the specified memory location. Indirect addressing may be used. The address of the memory location is computed by adding the contents of the X-register to *m* or to *m*,I. Note that indirect addressing (if specified) is performed first and then the address is indexed. The B-register and the X-register are not altered.

label	SAY	m [,I]	comments

Store A into memory indexed by Y. The contents of the A-register are copied into the specified memory location. Indirect addressing may be used. The address of the memory location is computed by adding the contents of the Y-register to *m* or to *m*,I. Note that indirect addressing (if specified) is performed first and then the address is indexed. The A-register and the Y-register are not altered.

label	SBY	m [,I]	comments

Store B into memory indexed by Y. The contents of the B-register are copied into the specified memory location. Indirect addressing may be used. The address of the memory location is computed by adding the contents of the Y-register to *m* or to *m*,I. Note that indirect addressing (if specified) is performed first and then the address is indexed. The B-register and the Y-register are not altered.

label	ADX	m [,I]	comments

Add memory to X. The contents of the specified memory location are algebraically added to the contents of the X-register. Indirect addressing may be used. The memory location is not altered.

label	ADY	m [,I]	comments

Add memory to Y. The contents of the specified memory location are algebraically added to the contents of the Y-register. Indirect addressing may be used. The memory location is not altered.

label	JLY	m [,I]	comments

Jump and load Y. Control transfers unconditionally to the specified memory location and the address P+2 is loaded into the Y-register. Indirect addressing may be used. This instruction is used for calling subroutines. The subroutines use the Y-register to access parameters and to return control (by way of the JPY instruction) to the calling program.

label	JPY	m	comments

Jump indexed by Y. Control transfers unconditionally to the specified memory location. Indirect addressing may *not* be used. The address of the memory location is computed by adding the contents of the Y-register to *m*. This instruction is used for returning control from subroutines to the calling program (assuming that they were entered by way of JLY instructions).

3-12. NO-OPERATION INSTRUCTION

When a no-operation is encountered in a program, no action takes place; the computer goes on to the next instruction. A full memory cycle is used in executing a no-operation instruction.

label	NOP	comments

A subroutine to be entered by a JSB instruction should have a NOP as the first statement. The return address can be stored in the location occupied by the NOP during execution of the program. A NOP statement causes the Assembler to generate a word of zero.

3-13. INPUT/OUTPUT, OVERFLOW, AND HALT

The input/output instructions allow the user to transfer data to and from an external device via a buffer, to enable or disable external interrupts, and to check the status of I/O devices and operations. A subset of these instructions permits checking for an arithmetic overflow condition.

Input/output instructions require the designation of a select code, sc, which indicates one of 64_{10} input/output channels or functions. Each channel consists of a control bit, a flag bit, and a buffer of up to 16 bits. Usually, the setting of the control bit starts operation of the device associated with the channel. The flag bit is set automatically when transmission between the device and the buffer is completed. Instructions are also available to test or clear the flag bit for the particular channel. If the interrupt system is enabled, setting of the flag causes program interrupt to occur; control transfers to the interrupt location related to the channel.

Note: When Memory Protect is enabled, execution of all I/O instructions except those which reference the switch register (select code 01) or the overflow bit *is prohibited.*

Expressions used to represent select codes (channel numbers) must have a value of less than 2^6. The value specifies the device or operation referenced. Instructions which transfer data between the A or B register and a buffer, access the switch register when sc = 1. The character C appended to such an instruction clears the overflow bit after the transfer from the switch register is complete. Unlike memory reference instructions, I/O instructions cannot use indirect links. The select code (sc) may be a label which was previously defined as an external symbol by an EXT pseudo-instruction. In such a case, the entry point referred to by the EXT pseudo-instruction must be an absolute value less than 64_{10} (any other value will change the instruction).

3-14. INPUT/OUTPUT

Assembly language programs normally perform I/O through calls to EXEC. Consult the appropriate RTE Programming and Operating manual for more information.

If the memory protect hardware option is present and enabled, it protects the operating system from alteration. The instructions of this section all cause memory protect violations to occur. They are included here for users who desire to write their own drivers.

To perform I/O, the software must set and clear the control and flag bits to communicate with the hardware devices. The installation and service manual of the I/O card being programmed should be consulted for the meaning of these bits for a specific device.

label	STC	sc [,C]	comments

Set I/O control bit for channel specified by sc. STC transfers or enables transfer of an element of data from an input device to the buffer or to an output device from the buffer. The exact function of the STC depends on the device; for the 2752A Teleprinter, an STC enables transfer of a series of bits. If sc = 1, this statement is treated as NOP. The C option clears the flag bit for the channel.

label	CLC	sc [,C]	comments

Clear I/O control bit for channel specified by sc. When the control bit is cleared, interrupt on the channel is disabled, although the flag may still be set by the device. If sc = 0, control bits for all channels are cleared to zero; all devices are disconnected. If sc = 1, this statement is treated as NOP.

label	LIA	sc [,C]	comments

Load into A the contents of the I/O buffer indicated by sc.

label	LIB	sc [,C]	comments

Load into B the contents of the I/O buffer indicated by sc.

label	MIA	sc [,C]	comments

Merge (inclusive "or") the contents of the I/O buffer indicated by sc into A.

label	MIB	sc [,C]	comments

Merge (inclusive "or") the contents of the I/O buffer indicated by sc into B.

label	OTA	sc [,C]	comments

Output the contents of A to the I/O buffer indicated by sc.

label	OTB	sc [,C]	comments

Output the contents of B to the I/O buffer indicated by sc.

label	STF	sc	comments

Set the flag bit of the channel indicated by sc. If sc = 0, STF enables the interrupt system. A sc code of 1 causes the overflow bit to be set.

label	CLF	sc	comments

Clear the flag bit to zero for the channel indicated by sc. If sc = 0, CLF disables the interrupt system. If sc = 1, the overflow bit is cleared to zero.

label	SFC	sc	comments

Skip the next instruction if the flag bit for channel sc is clear. If sc = 1, the overflow bit is tested. If sc = 0, the status of the interrupt system is tested.

label	SFS	sc	comments

Skip the next instruction if the flag bit for channel sc is set. If sc = 1, the overflow is tested. If sc = 0, the status of the interrupt system is tested.

Computer Museum

3-15 OVERFLOW

In addition to the use of a select code of 1, the overflow bit may be accessed by the following instructions:

label	CLO	comments

Clear the overflow bit.

label	STO	comments

Set overflow bit.

label	SOC	[C]	comments

Skip the next instruction if the overflow bit is clear. The C option clears the bit after the test is performed.

label	SOS	[C]	comments

Skip the next instruction if the overflow bit is set. The C option clears the bit after the test is performed.

The C option is identified by the sequence "space C space" following either "SOC" or "SOS". Any letter other than a "C" in this position will be treated as a comment.

3-16. HALT

label	HLT	{ [sc [,C]] / [C] }	comments

Halt the computer. The machine instruction word is displayed in the T-register. If the C option is used, the flag bit associated with channel sc is cleared.

If neither the select code nor the C option is used, the comments portion must be omitted.

3-17. EXTENDED ARITHMETIC UNIT (EAU)

If the computer on which the object program is to be run is an HP 21MX, this group of instructions may be used to increase the computer's overall efficiency.

If an HP 2114, HP 2115, HP 2116, or HP 2100 computer is being used, this group of instructions may be run if the computer contains an EAU.

The user specifies whether or not an EAU will be available via a parameter in the control statement (see paragraph 1-9). If an EAU will *not* be available, the instructions ASR, ASL, RRR, RRL, LSR, LSL, and SWP *cannot* be used in the source program (they will be flagged as errors) and the instructions MPY, DIV, DLD, and DST will result in calls to arithmetic subroutines (see paragraph 4-7).

label	MPY	{ m [,I] / lit }	comments

The MPY instruction multiplies the contents of the A-Register by the contents of m. The product is stored in registers B and A. B contains the sign of the product and the 15 most significant bits; A contains the least significant bits.

label	DIV	{ m [,I] / lit }	comments

The DIV instruction divides the contents of registers B and A by the contents of m. The quotient is stored in A and the remainder in B. Initially B contains the sign and the 15 most significant bits of the dividend; A contains the least significant bits.

label	DLD	{ m [,I] / lit }	comments

The DLD instruction loads the contents of locations m and m + 1 into registers A and B, respectively.

label	DST	m [,I]	comments

The DST instruction stores the contents of registers A and B in locations m and m + 1, respectively.

MPY, DIV, DLD, DST results in two machine words: a word for the instruction code and one for the operand.

The following seven instructions provide the capability to shift or rotate the B- and A-Registers n number of bit positions to the right or left, where $1 \leq n \leq 16$.

label	ASR	n	comments

The ASR instruction arithmetically shifts the B- and A-Registers right n bits. The sign bit (bit 15 of B) is extended.

label	ASL	n	comments

The ASL instruction arithmetically shifts the B- and A-Register left n bits. Zeroes are placed in the least significant bits. The sign bit (bit 15 of B) is unaltered. The overflow bit is set if bit 14 differs from bit 15 before each shift; otherwise, exit with overflow bit cleared.

label	RRR	n	comments

The RRR instruction rotates the B- and A-Registers right n bits.

label	RRL	n	comments

The RRL instruction rotates the B- and A-Registers left n bits.

label	LSR	n	comments

The LSR instruction logically shifts the B- and A-Registers right n bits. Zeroes are placed in the most significant bits.

label	LSL	n	comments

The LSL instruction logically shifts the B- and A-Registers left n bits. Place zeroes into the least significant bits.

	SWP	

Exchange the contents of the A- and B-Registers. The contents of the A-Register are shifted into the B-Register and the contents of the B-Register are shifted into the A-Register.

3-18. FLOATING POINT

The instructions in this group are used for performing arithmetic operations on floating point operands. The user specifies whether or not floating point machine instructions are available via a parameter in the control statement (see table 1-2). If the floating point machine instructions are *not* available, the instructions in this group result in calls to arithmetic subroutines (see paragraph 4-7). The Operand field may contain any relocatable expression or absolute expression resulting in a value of less than 2000_8.

label	FMP	{ m [,I] / =Fn }	comments

Multiply the two-word floating point quantity in registers A and B by the two-word floating point quantity in locations m and m+1 or the quantity defined by the literal. Store the two-word floating point product in registers A and B.

label	FDV	{ m [,I] / =Fn }	comments

Divide the two-word floating point quantity in registers A and B by the two-word floating point quantity in locations m and m+1 or the quantity defined by the literal. Store the two-word floating point quotient in A and B.

label	FAD	{ m [,I] / =FN }	comments

Add the two-word floating point quantity in registers A and B to the two-word floating point quantity in locations m and m+1 or the quantity defined by the literal. Store the two-word floating point sum in A and B.

label	FSB	{ m [,I] / =Fn }	comments

Subtract the two-word floating point quantity in m and m+1 or the quantity defined by the literal from the two-word floating point quantity in registers A and B and store the difference in A and B.

label	FIX	comments

Convert the floating-point number contained in the A- and B-registers to a fixed-point number. The result is returned in the A-register. After the operation is completed, the contents of the B-register are meaningless.

label	FLT	comments

Convert the fixed-point number contained in the A-register to a floating-point number. The result is returned in the A- and B-registers.

3-19. DYNAMIC MAPPING SYSTEM (21MX ONLY)

The basic addressing space of the HP 21MX Computer Series is 32,768 words, which is referred to as *logical* memory. The amount of memory actually installed in the computer system is referred to as *physical* memory. An HP 21MX Computer with the optional Dynamic Mapping System (DMS) has an addressing capability for one million words of memory. The DMS allows physical memory to be mapped into logical memory through the use of four dynamically alterable memory maps.

3-20. MEMORY ADDRESSING

The basic memory addressing scheme provides for addressing 32 pages of logical memory, each of which consists of 1,024 words. This memory is addressed through a 15-bit memory address bus shown in figure 3-1. The upper 5 bits of this bus provide the page address and the lower 10 bits provide the relative word address within the page.

The Memory Expansion Module (MEM), which is part of the DMS option, converts the 5-bit page address into a 10-bit page address and thereby allows 1,024 (2^{10}) *pages* to be addressed. This conversion is accomplished by allowing the original 5-bit address to identify one of the 32 registers within a "memory map." Each of these map registers contains the new user-specified 10-bit page address. This new page address is combined with the original 10-bit relative address to form a 20-bit memory address bus as shown in figure 3-2.

3-21. STATUS AND VIOLATION REGISTERS

The MEM also includes a status register and a violation register. As shown in table 3-1, the MEM status register contents enable the programmer to determine whether the MEM was enabled or disabled at the time of the last interrupt and the address of the base page fence. The MEM violation register contents enable the programmer to determine whether a fault occurred in the hardware or the software so that the proper corrective steps may be taken. Refer to table 3-2.

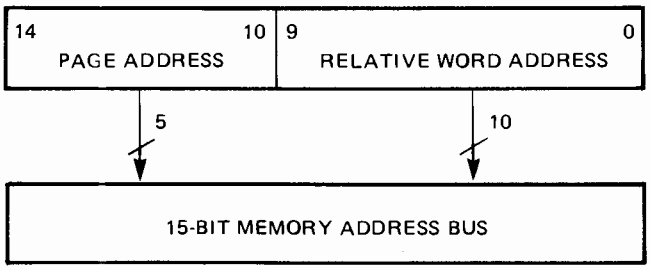


Figure 3-1. Basic Memory Addressing Scheme

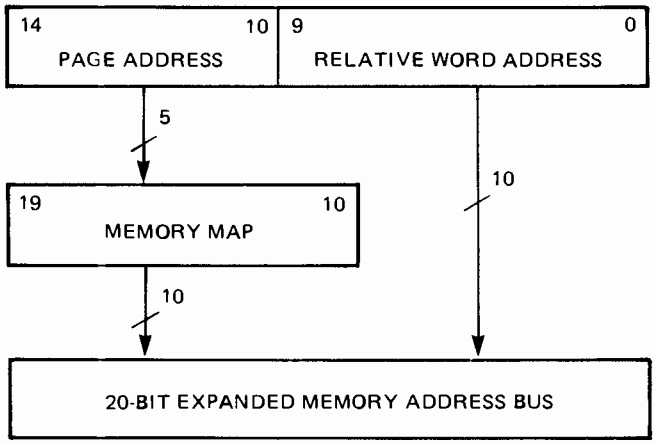


Figure 3-2. Expanded Memory Addressing Scheme

Table 3-1. MEM Status Register Format

BIT	SIGNIFICANCE
15	0 = MEM disabled at last interrupt 1 = MEM enabled at last interrupt
14	0 = System map selected at last interrupt 1 = User map selected at last interrupt
13	0 = MEM disabled currently 1 = MEM enabled currently
12	0 = System map selected currently 1 = User map selected currently
11	0 = Protected mode disabled currently 1 = Protected mode enabled currently
10	Portion mapped*
9	Base page fence bit 9
8	Base page fence bit 8
7	Base page fence bit 7
6	Base page fence bit 6
5	Base page fence bit 5
4	Base page fence bit 4
3	Base page fence bit 3
2	Base page fence bit 2
1	Base page fence bit 1
0	Base page fence bit 0

*Bit 10	Mapped Address (M)
0	Fence $\leqslant$ M $<$ 2000_8
1	1 $<$ M $<$ Fence

3-22. MAP SEGMENTATION

All registers within the memory map are dynamically alterable. The MEM includes four separate memory maps: the User Map, System Map, and two Dual-Channel Port Controller (DCPC) Maps. See figure 3-3. These maps are addressed as a contiguous register block.

3-23. POWER FAIL CHARACTERISTICS

A power failure automatically enables the System Map, and a minimum of 500 microseconds is assured the programmer for executing a power fail routine. Since all maps

Table 3-2. MEM Violation Register Format

BIT	SIGNIFICANCE
15	Read violation*
14	Write violation*
13	Base page violation*
12	Privileged instruction violation*
11	Reserved
10	Reserved
9	Reserved
8	Reserved
7	0 = ME bus disabled at violation 1 = ME bus enabled at violation
6	0 = MEM disabled at violation 1 = MEM enabled at violation
5	0 = System map enabled at violation 1 = User map enabled at violation
4	Map address bit 4
3	Map address bit 3
2	Map address bit 2
1	Map address bit 1
0	Map address bit 0

*Significant when associated bit is set.

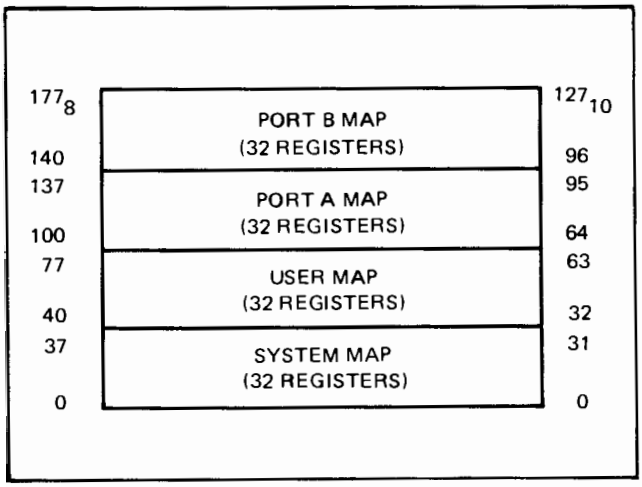


Figure 3-3. Map Segmentation

are disabled and none are considered valid upon the restoration of power, the power fail routine should include instructions to save as many maps as desired.

3-24. PROTECTED MODE

The protected mode of operation is a program state created by the Dynamic Mapping System. The protected mode is entered by executing an STC 05 instruction and is exited by the CPU acknowledging an interrupt. The protected mode reserves a block of memory and prevents access to this block by other users.

3-25. MEM VIOLATION

An interrupt request which attempts to access the protected block of memory (while in the protected mode) will cause a MEM violation.

3-26. DYNAMIC MAPPING SYSTEM INSTRUCTIONS

If the computer on which the object program is to be run includes a Dynamic Mapping System, the following group of instructions may be used.

label	DJP	m [,I]	comments

Disable MEM and jump. This instruction disables the translation and protection features of the MEM hardware. Prior to disabling, the P-register is set to the effective memory address. As a result of executing this instruction, normal I/O interrupts are held off until the first opportunity following the fetch of the next instruction, unless three or more levels of indirect addressing are used.

This instruction will normally generate a MEM violation when executed in the protected mode. In this case, the status of the MEM is not affected and the jump will not occur; however, if the System map is enabled, the instruction is allowed. If none of the maps are enabled, the instruction defaults to JMP*+1,I.

label	DJS	m [,I]	comments

Disable MEM and jump to subroutine. This instruction disables the translation and protection features of the MEM hardware. Prior to disabling, the P-register is set one count past the effective memory address. The return address is written into the location specified by *m* [,I] . As a result of executing this instruction, normal I/O interrupts are held off until the first opportunity following the fetch of the next instruction, unless three or more levels of indirect addressing are used.

This instruction will normally generate a MEM violation when executed in the protected mode. In this case, the status of the MEM is not affected and the jump will not occur; however, if the System map is enabled, the instruction is allowed.

label	JRS	{addr1 [,I] / literal}	addr2 [,I]	comments

Jump and restore status. *addr1* contains the address of the status word memory location, *literal* specifies the status word, and *addr2* contains the jump address.

This instruction causes the status of MEM to be restored as indicated by bits 15 and 14 of the status word. Only bits 15 and 14 of the status word are used; the remaining bits (13-0) of the status word are ignored. Bits 15 and 14 restore the MEM status as follows:

Bit 15 = 0 MEM is disabled

Bit 15 = 1 MEM is enabled

Bit 14 = 0 System map is selected

Bit 14 = 1 User map is selected.

As a result of executing this instruction, normal I/O interrupts are held off until the first opportunity following the fetch of the next instruction, unless three or more levels of indirect addressing are used.

This instruction will normally generate a MEM violation when executed in the protected mode. In this case, the status of the MEM is not affected and the jump will not occur; however, if the System map is enabled, the instruction is allowed.

label	LFA	comments

Load fence from A. This instruction loads the contents of the A-register into the base page fence register. (The base page fence register contains the "fence" address, which specifies the address where reserved (mapped) memory begins. Attempts to access memory at any address below this fence will not be allowed.) Bits 9 through 0 of the A-register specify the address in page zero where shared (unmapped) memory is separated from reserved (mapped) memory. Bit 10 is used as follows to specify which portion is mapped:

Bit 10	Mapped Address (M)
0	Fence $\leq$ M $<$ 2000_8
1	1 $<$ M $<$ Fence

This instruction will always generate a MEM violation when executed in the protected mode. In this case, the fence is not altered. However, if the System map is enabled, the instruction is allowed in protected mode.

label	LFB	comments

Load fence from B. This instruction loads the contents of the B-register into the base page fence register. Bits 9 through 0 of the B-register specify the address in page zero where shared (unmapped) memory is separated from reserved (mapped) memory. Bit 10 is used as follows to specify which portion is mapped:

Bit 10	Mapped Address (M)
0	Fence $\leq$ M $<$ 2000_8
1	1 $<$ M $<$ Fence

This instruction will always generate a MEM violation when executed in the protected mode. In this case, the fence is not altered. However, if the System map is enabled, the instruction is allowed in protected mode.

label	MBF	comments

Move bytes from alternate map. This instruction moves a string of bytes using the alternate program map for source reads and the currently enabled map for destination writes. (The alternate map is the map which is not enabled. For example, if the system map is enabled, the User map is the alternate map and vice versa.) The A-register contains the source byte address and the B-register contains the destination byte address. The X-register contains the octal number of bytes to be moved. Both the source and destination byte address must begin on even word boundaries.

This instruction is interruptible on an even number of byte transfers, thus maintaining the even word boundaries in the A- and B-registers.

The interrupt routine is expected to save and restore the current contents of the A-, B-, and X-registers to allow continuation of the instruction at the next entry. When the byte string move is completed, the X-register will always be zero and the A- and B-registers will contain their original value incremented by the number of bytes moved.

This instruction can cause a MEM violation only if read or write protection rules are violated. (For example, if an attempt is made to write to the reserved (mapped) section of memory.)

label	MBI	comments

Move bytes into alternate map. This instruction moves a string of bytes using the currently enabled map for source reads and the alternate program map for destination writes. The A-register contains the source byte address and the B-register contains the destination byte address. The X-register contains the octal number of bytes to be moved. Both the source and destination byte addresses must begin on even word boundaries.

This instruction is interruptible on an even number of byte transfers, thus maintaining the even word boundaries in the A- and B-registers. The interrupt routine is expected to

save and restore the current contents of the A-, B-, and X-registers to allow continuation of the instruction at the next entry. When the byte string move is completed, the X-register will always be zero and the A- and B-registers will contain their original value incremented by the number of bytes moved.

This instruction will always cause a MEM violation when executed in the protected mode and no bytes will be transferred.

label	MBW	comments

Move bytes within alternate map. This instruction moves a string of bytes with both the source and destination addresses established through the alternate program map. The A-register contains the source byte address and the B-register contains the destination byte address. The X-register contains the octal number of bytes to be moved. Both the source and destination byte addresses must begin on even word boundaries.

This instruction is interruptible on an even number of byte transfers, thus maintaining the even word boundaries in the A- and B-registers.

The interrupt routine is expected to save and restore the current contents of the A-, B- and X-registers to allow continuation of the instruction at the next entry. When the byte string move is completed, the X-register will always be zero and the A- and B-registers will contain their original value incremented by the number of bytes moved.

This instruction will always cause a MEM violation when executed in the protected mode and no bytes will be transferred.

label	MWF	comments

Move words from alternate map. This instruction moves a string of words using the alternate program map for source reads and the currently enabled map for destination writes. The A-register contains the source address and the B-register contains the destination address. The X-register contains the octal number of words to be moved.

This instruction is interruptible. The interrupt routine is expected to save and restore the current contents of the A-B-, and X-registers to allow continuation of the instruction at the next entry. When the word string move is completed, the X-register will always be zero and the A- and B-registers will contain their original value incremented by the number of words moved.

This instruction can cause a MEM violation only if read and write protection rules are violated.

label	MWI	comments

Move words into alternate map. This instruction moves a string of words using the currently enabled map for source reads and the alternate program map for destination writes. The A-register contains the source address and the B-register contains the destination address. The X-register contains the octal number of words to be moved.

This instruction is interruptible. The interrupt routine is expected to save and restore the current contents of the A-, B-, and X-registers to allow continuation of the instruction at the next entry. When the word string move is completed, the X-register will always be zero and the A- and B-registers will contain their original value incremented by the number of words moved.

This instruction will always cause a MEM violation when executed in the protected mode and no words will be transferred.

label	MWW	comments

Move words within alternate map. This instruction moves a string of words with both the source and destination addresses established through the alternate program map. The A-register contains the source address and the B-register contains the destination address. The X-register contains the octal number of words to be moved.

This instruction is interruptible. The interrupt routine is expected to save and restore the current contents of the A-, B-, and X-registers to allow continuation of the instruction at the next entry. When the word string move is completed, the X-register will always be zero and the A- and B-registers will contain their original value incremented by the number of words moved.

This instruction will always cause a MEM violation when executed in the protected mode and no words will be transferred.

label	PAA	comments

Load/store Port A map per A. This instruction transfers the 32 Port A map registers to or from memory. If bit 15 of the A-register is clear, the Port A map is *loaded* from memory starting from the address specified in bits 14-0 of the A-register. If bit 15 of the A-register is set, the Port A map is *stored* into memory starting at the address specified in bits 14-0 of the A-register. When the load/store operation is complete, the A-register will be incremented by 32 to allow multiple map instructions.

An attempt to load any map register when in the protected mode will cause a MEM violation. An attempt to store the Port A map is allowed within the constraints of write protected memory.

label	PAB	comments

Load/store Port A map per B. This instruction transfers the 32 Port A registers to or from memory. If bit 15 of the B-register is clear, the Port A map is *loaded* from memory

starting from the address specified in bits 14-0 of the B-register. If bit 15 of the B-register is set, the Port A map is *stored* into memory starting at the address specified in bits 14-0 of the B-register. When the load/store operation is complete, the B-register will be incremented by 32 to allow multiple map instructions.

An attempt to load any map register when in the protected mode will cause a MEM violation. An attempt to store the Port A map is allowed within the constraints of write protected memory.

label	PBA	comments

Load/store Port B map per A. This instruction transfers the 32 Port B registers to or from memory. If bit 15 of the A-register is clear, the Port B map is *loaded* from memory starting from the address specified in bits 14-0 of the A-register. If bit 15 of the A-register is set, the Port B map is *stored* into memory starting at the address specified in bits 14-0 of the A-register. When the load/store operation is complete, the A-register will be incremented by 32 to allow multiple map instructions.

An attempt to load any map register when in the protected mode will cause a MEM violation. An attempt to store the Port B map is allowed within the constraints of write protected memory.

label	PBB	comments

Load/store Port B map per B. This instruction transfers the 32 Port B map registers to or from memory. If bit 15 of the B-register is clear, the Port B map is *loaded* from memory starting from the address specified in bits 14-0 of the B-register. If bit 15 of the B-register is set, the Port B map is *stored* into memory starting at the address specified in bits 14-0 of the B-register. When the load/store operation is complete, the B-register will be incremented by 32 to allow multiple map instructions.

An attempt to load any map register when in the protected mode will cause a MEM violation. An attempt to store the Port B map is allowed within the constraints of the write protected memory.

label	RSA	comments

Read status register into A. This instruction reads the contents of the MEM status register into the A-register. This instruction can be executed at any time. The format of the MEM status register is given in table 3-1.

label	RSB	comments

Read status register into B. This instruction reads the contents of the MEM status register into the B-register and can be executed at any time. The format of the MEM status register is shown in table 3-1.

label	RVA	comments

Read violation register into A. This instruction reads the contents of the MEM violation register into the A-register and can be executed at any time. The format of the MEM violation register is shown in table 3-2.

label	RVB	comments

Read violation register into B. This instruction reads the contents of the MEM violation register into the B-register and can be executed at any time. The format of the MEM violation register is shown in table 3-2.

label	SJP	m [,I]	comments

Enable System map and jump. This instruction causes the MEM hardware to use the set of 32 map registers, referred to as the System map, for translating all programmed memory references. Prior to enabling the System map, the P-register is set to the effective memory address. As a result of executing this instruction, normal I/O interrupts are held off until the first opportunity following the fetch of the next instruction, unless three or more levels of indirect addressing are used.

This instruction will normally generate a MEM violation when executed in the protected mode. In this case, the status of the MEM is not affected and the jump will not occur; however, if the System map is enabled, the instruction is allowed and effectively executes a JMP *+1, I.

label	SJS	m [,I]	comments

Enable System map and jump to subroutine. This instruction causes the MEM hardware to use the set of 32 map registers, referred to as the System map, for translating all programmed memory references. Prior to enabling the System map, the P-register is set one count past the effective memory address. The return address is written into the location specified by *m* [,I]. As a result of executing this instruction, normal I/O interrupts are held off until the first opportunity following the fetch of the next instruction, unless three or more levels of indirect addressing are used.

This instruction will normally generate a MEM violation when executed in the protected mode. In this case, the status of the MEM is not affected and the jump will not occur; however, if the System map is enabled, the instruction is allowed and effectively executes a JSB *+1,I.

label	SSM	m [,I]	comments

Store status register in memory. This instruction stores the 16-bit contents of the MEM status register into the addressed memory location. The status register contents are not altered. This instruction is used in conjunction with the

JRS instruction to allow easy processing of interrupts, which always select the System map (if the MEM is enabled). The format of the MEM status register is listed in table 3-1.

This instruction can cause a MEM violation only if write protection rules are violated.

label	SYA	comments

Load/store System map per A. This instruction transfers the 32 System map registers to or from memory. If bit 15 of the A-register is clear, the System map is *loaded* from memory starting from the address specified in bits 14-0 of the A-register. If bit 15 of the A-register is set, the System map is *stored* into memory starting at the address specified in bits 14-0 of the A-register. When the load/store operation is complete, the A-register will be incremented by 32 to allow multiple map instructions.

Note: If not in the protected mode, the MEM provides no protection against altering the contents of maps while they are currently enabled.

An attempt to load any map in the protected mode will cause a MEM violation. An attempt to store the System map is allowed within the constraints of write protected memory.

label	SYB	comments

Load/store System map per B. This instruction transfers the 32 System map registers to or from memory. If bit 15 of the B-register is clear, the system map is *loaded* from memory starting from the address specified in bits 14-0 of the B-register. If bit 15 of the B-register is set, the System map is *stored* into memory starting at the address specified in bits 14-0 of the B-register. When the load/store operation is complete, the B-register will be incremented by 32 to allow multiple map instructions.

Note: If not in the protected mode, the MEM provides no protection against altering the contents of maps while they are currently enabled.

An attempt to load any map in the protected mode will cause a MEM violation. An attempt to store the System map is allowed within the constraints of write protected memory.

label	UJP	m [,I]	comments

Enable User map and jump. This instruction causes the MEM hardware to use the set of 32 map registers, referred to as the User map, for translating all programmed memory references. Prior to enabling the User map, the P-register is set to the effective memory address. As a result of executing this instruction, normal I/O interrupts are held off until the first opportunity following the fetch of the next instruction, unless three or more levels of indirect addressing are used. If the User map is already enabled, the instruction defaults to JMP *+1,I.

This instruction will normally generate a MEM violation when executed in the protected mode. In this case, the status of the MEM is not affected and the jump will not occur; however, if the System map is enabled, the instruction is allowed.

label	UJS	m [,I]	comments

Enable User map and jump to subroutine. This instruction causes the MEM hardware to use the set of 32 map registers, referred to as the User map, for translating all programmed memory references. Prior to enabling the User map, the P-register is set one count past the effective memory address. The return address is written into the location specified by *m* [,I]. As a result of executing this instruction, normal I/O interrupts are held off until the first opportunity following the fetch of the next instruction, unless three or more levels of indirect addressing are used. If the User map is already enabled, the instruction defaults to JMP *+1,I.

This instruction will normally generate a MEM violation when executed in the protected mode. In this case, the status of the MEM is not affected and the jump will not occur; however, if the System map is enabled, the instruction is allowed.

label	USA	comments

Load/store User map per A. This instruction transfers the 32 User map registers to or from memory. If bit 15 of the A-register is clear, the User map is *loaded* from memory starting from the address specified in bits 14-0 of the A-register. If bit 15 of the A-register is set, the User map is *stored* into memory starting at the address specified in bits 14-0 of the A-register. When load/store operation is complete, the A-register will be incremented by 32 to allow multiple instructions.

Note: If not in the protected mode, the MEM provides no protection against altering the contents of maps while they are currently enabled.

An attempt to load any map in the protected mode will cause a MEM violation. An attempt to store the User map is allowed within the constraints of write protected memory.

label	USB	comments

Load/store User map per B. This instruction transfers the 32 User map registers to or from memory. If bit 15 of the B-register is clear, the User map is *loaded* from memory starting from the address specified in bits 14-0 of the B-register. If bit 15 of the B-register is set, the User map is

stored into memory starting at the address specified in bits 14-0 of the B-register. When the load/store operation is complete, the B-register will be incremented by 32 to allow multiple map instructions.

Note: If not in the protected mode, the MEM provides no protection against altering the contents of maps while they are currently enabled.

An attempt to load any map in the protected mode will cause a MEM violation. An attempt to store the User map is allowed within the constraints of write protected memory.

label	XCA	m [,I]	comments

Cross compare A. This instruction compares the contents of the A-register with the contents of the addressed memory location. If the two 16-bit words are not identical, the next instruction is skipped; i.e., the P-register advances two counts instead of one count. If the two words are identical, the next instruction is executed. Neither the A-register nor memory cell contents are altered.

This instruction uses the alternate program map for the read operation. If neither the System map nor the User map is enabled (i.e., MEM is disabled), then a compare directly with physical memory occurs. This instruction will cause a MEM violation only if read protection rules are violated.

label	XCB	m [,I]	comments

Cross compare B. This instruction compares the contents of the B-register with the contents of the addressed memory location. If the two 16-bit words are not identical, the next instruction is skipped; i.e., the P-register advances two counts instead of one count. If the two words are identical, the next instruction is executed. Neither the B-register contents nor memory cell contents are altered.

This instruction uses the alternate map for the read operation. If neither the System map nor the User map is enabled (i.e., MEM is disabled), then a direct compare with physical memory occurs.

This instruction will cause a MEM violation only if read protection rules are violated.

label	XLA	m [,I]	comments

Cross load A. This instruction loads the contents of the specified memory address into the A-register. The contents of the memory cell are not altered.

This instruction uses the alternate program map to fetch the operand. If the MEM is currently disabled, then a load directly from physical memory occurs.

This instruction will cause a MEM violation only if read protection rules are violated.

label	XLB	m [,I]	comments

Cross load B. This instruction loads the contents of the specified memory address into the B-register. The contents of the memory cell are not altered.

This instruction uses the alternate program map to fetch the operand. If the MEM is currently disabled, then a load directly from physical memory occurs.

This instruction will cause a MEM violation only if read protection rules are violated.

label	XMA	comments

Transfer maps internally per A. This instruction transfers a copy of the entire contents (32 map registers) of the System map or the User map to the Port A map or the Port B map as determined by the control word in the A-register, as follows:

Bit No.	Significance
15	0 = System map 1 = User map
0	0 = Port A map 1 = Port B map

(Bits 14-1 are ignored)

This instruction will always cause a MEM violation when executed in the protected mode.

label	XMB	comments

Transfer maps internally per B. This instruction transfers a copy of the entire contents (32 map registers) of the System map or the User map to the Port A map or the Port B map as determined by the control word in the B-register, as follows:

Bit No.	Significance
15	0 = System map 1 = User map
0	0 = Port A map 1 = Port B map

(Bits 14-1 are ignored)

This instruction will always generate a MEM violation when executed in the protected mode.

The IFN and IFZ pseudo instructions cause the inclusion of instructions in a program provided that either an "N" or "Z", respectively, is specified as a parameter for the ASMB control statement.† The IFN or IFZ instruction precedes the set of statements that are to be included. The pseudo instruction XIF serves as a terminator. If XIF is omitted, END acts as a terminator to both the set of statements and the assembly.

```
        | IFN | comments
          .
          .
          .
        XIF
```

All source language statements appearing between the IFN and the XIF pseudo instructions are included in the program if the character "N" is specified on the ASMB control statement.

All source language statements appearing between the IFZ and XIF pseudo instructions are included in the program if the character "Z" is specified on the ASMB control statement.

```
        | IFZ | comments
          .
          .
          .
        XIF
```

When the particular letter is not included on the control statement, the related set of statements appears on the Assembler output listing but is not assembled.

Any number of IFN-XIF and IFZ-XIF sets may appear in a program, however, they may not overlap. An IFZ or IFN intervening between an IFZ or IFN and the XIF terminator results in a diagnostic being issued during compilation; the second pseudo instruction is ignored.

Both IFN-XIF and IFZ-XIF pseudo instructions may be used in the program; however, only one type will be selected in a single assembly. Therefore, if both characters "N" and "Z" appear in the control statement, the character which is listed last will determine the set of coding that is to be assembled. Some examples are shown in figures 4-4 and 4-5.

In figure 4-4, the program TRAVL will perform computations involving either or neither CAR or PLANE considerations depending on the presence or absence of Z or N parameters in the Control Statement.

In figure 4-5, the program WAGES computes a weekly wage value. Overtime consideration will be included in the program if "Z" is included in the parameters of the Control Statement.

```
        NAM TRAVL
         .
         .
         .
        IFZ
        LDA CAR
        CMA,SZA
        JMP NO.GO
        LDA MILES
        DIV SPEED
        STA GAS
        XIF
         .
         .
         .
        IFN
        LDA PLANE
        CMA,SZA
        JMP NO.GO
        LDA TIME
        CPA COST
        XIF
NO.GO   HLT 77
         .
         .
         .
        END
```

Figure 4-4. IFN/XIF and IFZ/XIF Example

```
        NAM WAGE
         .
         .
         .
        JSB HOUR
        MPY TIME1
        IFZ
        JSB OVTIM
        MPY TIME2
         .
         .
         .
TIME1   DEC 40
TIME2   BSS 1
        END
```

Figure 4-5. IFZ/XIF Example

†See "Assembly Options" in Section I of this manual.

The REP pseudo instruction causes the repetition of the statement immediately following it a specified number of times.

label	REP	n	comments

The statement following the REP in the source program is repeated n times. The n may be any absolute expression. Comment lines (indicated by an asterisk in character position 1) are not repeated by REP. If a comment follows a REP instruction, the comment is ignored and the instruction following the comment is repeated.

A label specified in the REP pseudo instruction is assigned to the first repetition of the statement. A label should not be part of the instruction to be repeated; it would result in a doubly defined symbol error.

Example:

```
        CLA
TRIPL   REP   3
        ADA   DATA
```

The above source code would generate the following:

```
        CLA                 Clear the A-Register;
                            the content of DATA is
TRIPL   ADA   DATA          tripled and stored in the
        ADA   DATA          A-Register.
        ADA   DATA
```

Example:

```
FILL    REP   100B
        NOP
```

The example above loads 100_8 memory locations with the NOP instruction. The first location is labeled FILL.

Example:

```
        REP   2
        MPY   DATA
```

The above source code would generate the following:

```
        MPY   DATA
        MPY   DATA
```

	END	[m]	comments

This statement terminates the program; it marks the physical end of the source language statements. The Operand field, m, may contain a name appearing as a statement label in the current program or it may be blank. If a name is entered, it identifies the location to which the loader transfers control after a relocatable program is loaded.

If the Operand field is blank, the Comments field must be blank also, otherwise, the Assembler attempts to interpret the first five characters of the comments as the transfer address symbol.

The label field of the END statement is ignored.

4-2. OBJECT PROGRAM LINKAGE

Linking pseudo instructions provides a means for communication between a main program and its subroutines or among several subprograms that are to be run as a single program. These instructions may be used only in a relocatable program.

The Label field of this class is ignored in all cases. The Operand field is usually divided into many subfields, separated by commas. In the case of the COM pseudo instruction, the first space not preceded by a comma or a left parenthesis terminates the entire field.

Computer Museum

	COM	$name_1[(size_1)]$ $[,name_2[(size_2)]$, . . . , $name_n[(size_n)]]$	comments

COM reserves a block of storage locations that may be used in common by several subprograms. Each name identifies a segment of the block for the subprogram in which the COM statement appears. The sizes are the number of words allotted to the related segments. The size is specified as an octal or decimal integer. If the size is omitted, it is assumed to be one.

Any number of COM statements may appear in a subprogram. Storage locations are assigned contiguously; the length of the block is equal to the sum of the lengths of all segments named in all COM statements in the subprogram.

To refer to the common block, other subprograms must also include a COM statement. The segment names and sizes may be the same or they may differ. Regardless of the names and sizes specified in the separate subprograms, there is only one common block for the combined set. It has the same relative origin; the content of the nth word or common storage is the same for all subprograms. An example is shown in figure 4-6.

The LDA instructions in the two subprograms each refer to the same location in common storage, location 7.

```
PROG1 COM ADDR1(5),ADDR2(10),ADDR3(10)
        .
        .
        .
      LDA ADDR2+1    PICK UP SECOND WORD OF SEGMENT
        .            ADDR2+1
        .
      END
        .
        .
PROG2 COM AAA(2),AAB(2),AAC,AAD(20)
        .
        .
      LDA AAD+1      PICK UP SECOND WORD OF SEGMENT
                     ADD+1.
```

Organization of common block:

PROG1 name	PROG2 name	Common Block
ADDR1	AAA	(location 1)
		(location 2)
	AAB	(location 3)
		(location 4)
	AAC	(location 5)
ADDR2	AAD	(location 6)
		(location 7)
		(location 8)
		(location 9)
		(location 10)
		(location 11)
		(location 12)
		(location 13)
		(location 14)
		(location 15)
ADDR3		(location 16)
		(location 17)
		(location 18)
		(location 19)
		(location 20)
		(location 21)
		(location 22)
		(location 23)
		(location 24)
		(location 25)

Figure 4-6. COM Examples

The segment names that appear in the COM statements can be used in the Operand fields of DEF, ABS, EQU, ENT or any memory reference statement; they may not be used as labels elsewhere in the program.

The loader establishes the origin of the common block; the origin cannot be set by the ORG pseudo instruction. All references to the common area are relocatable.

Two or more subprograms may declare common blocks that differ in size. The subprogram that defines the largest block must be the first submitted for loading.

	ENT	$name_1$ [,$name_2$,...,$name_n$]	comments

ENT defines entry points to the program or subprogram. Each name is a symbol that is assigned as a label for some machine operation in the program. Entry points allow another subprogram to refer to this subprogram. All entry points must be defined in the program.

Symbols appearing in an ENT statement may not also appear in an EXT statement in the same subprogram. Labels defined as absolute by EQU statements or defined by COM statements may be declared as entry points.

	EXT	$name_1$ [,$name_2$,...,$name_n$]	comments

This instruction designates labels in other subprograms that are referenced in this subprogram. The symbols must be defined as entry points by the other subprograms.

The symbols defined in the EXT statement may appear in memory reference statements, certain I/O statements or EQU or DEF pseudo instructions. An external symbol may be used with a + or − offset or specified as indirect. References to external locations are processed by the loader as indirect addresses linked through the base page or in some cases through a current page link.

Symbols appearing in EXT statements may not also appear in ENT or COM statements in the same subprogram. The label field is ignored. Examples of the use of EXT and ENT are shown in figures 4-7 through 4-10.

```
PROGA NOP
      LDA SAMD        SAMD AND SAND ARE REFERENCED IN
       .              PROGA, BUT ARE ACTUALLY
       .              LOCATIONS IN PROGB.
       .
      JMP SAND
      EXT SAMD,SAND
      ENT PROGA
      END
       .
       .
PROGB NOP
       .
       .
SAMD  OCT 767
SAND  STA SAMD
       .
       .

      ENT SAMD,SAND
       .
       .
      JSB PROGA
       .
       .
      EXT PROGA
       .
       .
      END
```

Figure 4-7. ENT/EXT Examples

```
EXT BUF,PTR
 .
 .
 .
LDA BUF+1    EXTERNAL WITH + OR - OFFSET.
STA PTR,I    EXTERNAL INDIRECT.
```

Figure 4-8. EXT with Offset

```
        ENT CHAN,CMLBL
         .
         .
         .
CHAN    EQU 12B
        COM CMLBL  (20)
```

Figure 4-9. ENT in COMmon and ENT Defining
An External I/O Reference

```
ASMB,R,B,L
      NAM MAIN
*
*     DECLARE CHAN1,CHAN2 AS ENTRY POINTS
*
      ENT CHAN1,CHAN2
      EXT OUTPT,INPUT

START JSB INPUT INPUT A CHARACTER
      JSB OUTPT OUTPUT TO DEVICE 2
      LIA 1B    READ SWITCH REGISTER
      SSA       IS BIT 15 ON?
      HLT 55B   YES, HALT
      JMP START DO ANOTHER ONE
*
*     DEFINE THE I/O CHANNELS FOR THE DRIVERS INPUT,OUTPT BY
*     SETTING THE LABELS CHAN1,CHAN2 EQUIVALENT TO THE ABSOLUTE
*     LOCATIONS 10,11.

CHAN1 EQU 10B
CHAN2 EQU 11B
      END START
```

```
ASMB,R,B,L
      NAM IOPRG
*     SUBROUTINE ENTRY POINTS
      ENT INPUT,OUTPT
*     DECLARE I/O CHANNELS TO BE EXTERNAL
      EXT CHAN1,CHAN2
*
*     INPUT     SUBROUTINE
*
INPUT NOP
      STC CHAN1,C    SET CONTROL ON CHANNEL 1
      SFS CHAN1
      JMP *-1   WAIT ON FLAG
      LIA CHAN1. LOAD WORD
      JMP INPUT,I    RETURN
*
*     OUTPUT SUBROUTINE
*
OUTPT NOP
      OTA CHAN2 OUTPUT WORD
      STC CHAN2,C    STROBE TO DEVICE
      SFS CHAN2
      JMP *-1
      JMP OUTPT,I    RETURN

      END
```

Figure 4-10. EXT, ENT for I/O Channel

The RPL pseudo instruction is used to define a code replacement record for the RTE system generator or RTE relocating loader.

label	RPL	m	comments

The instructions to be replaced must be of the form = JSB SUB where SUB is an external reference. The JSB SUB will be replaced by the octal value of the RPL definition whenever it is encountered by the generator or loader. Examples are shown in figure 4-11.

```
.FAD RPL   105000B
IFIX RPL   105100B
```

```
EXT    .FAD
  :
JSB    .FAD
  :
JSB    IFIX
  :
```

Figure 4-11. Label RPL Octal Value

The relocation of the program would result in the following:

```
  :
105000        Note that the instruction value is 105000
              instead of 114XXX.
  :
105100        Note that the instruction value is 105100
              instead of 114XXX.
  :
```

4-3. ADDRESS AND SYMBOL DEFINITION

The pseudo operations in this group assign a value, a word address, or a byte address to a symbol which is used as an operand elsewhere in the program.

label	DEF	m [,I]	comments

The address definition statement generates one word of memory as a 15-bit address which may be used as the object of an indirect address found elsewhere in the source program. The symbol appearing in the label is that which is referenced; it appears in the Operand field of a Memory Reference instruction.

The operand field of the DEF statement may be any positive expression in an absolute program; in a relocatable program it may be a relocatable expression or an absolute expression with a value of less than 2000_8. Symbols that do appear in the Operand field may appear as operands of EXT or COM statements, in the same subprogram and as entry points in other subprograms.

The expression in the Operand field may itself be indirect and make reference to another DEF statement elsewhere in the source program. Some examples are shown in figure 4-12.

The DEF statement provides the necessary flexibility to perform address arithmetic in programs which are to be assembled in relocatable form. *Relocatable programs should not modify the operand of a memory reference instruction.* Figure 4-13 illustrates what *not* to do. If TBL and LDTBL are in different pages, the Loader processes TBL as an indirect address linked through the base page. The ISZ erroneously increments the Loader-provided link to the base page rather than the value of TBL. Assuming that the loader assigns the absolute locations shown in figure 4-14, the ISZ will index the contents of location 2000_8 which is a LDA 700,I, and change it to LDA 701,I. Now we will use whatever happens to be in 701 rather than the link we intended to use which is in 700. We change the *link* instead of its *contents*.

```
LDTBL LDA TBL
        .
        .
        .
      ISZ LDTBL
        .
        .
        .
TBL   BSS 100
```

Figure 4-13. Example of Incorrect Address Modification

```
        NAM PROGN      ZERO-RELATIVE START OF PROGRAM.
        EXT SINE,SQRT
        COM SCMA(20),SCMB(50)
         .
         .
        JSB SINE       EXECUTE SINE ROUTINE
         .
         .
        LDA XCMA,I     PICK UP COMMON WORD INDIRECTLY.
         .
         .
XCMA    DEF SCMA       SCMA IS A 15-BIT ADDRESS.
         .
         .
        JSB XSQ,I      GET SQUARE ROOT USING TWO-LEVEL
XSQ     DEF XSQR,I     INDIRECT ADDRESSING.
         .
         .
XSQR    DEF SQRT       SQRT IS A 15-BIT ADDRESS.
        END PROGN
```

Figure 4-12. DEF Examples

INSTRUCTION			PAGE	ABSOLUTE LOCATION OF CODE	OPCODE	OPERAND (PAGE) & LOCATION
(Loader-assigned indirect link on base page)			(0)	(700)	DEF	4000
					.	
					.	
LDTBL	LDA	TBL	(1)	(2000)	LDA	(0) 700 (I)
					.	
					.	
	ISZ	LDTBL	(1)	(3000)	ISZ	(1) 2000
					.	
					.	
TBL	BSS	100	(2)	(4000)	BSS	

Figure 4-14. Loader-Assigned Locations for Figure 4-8

The example shown in figure 4-15 assures correct address modification during program execution. Assume that the sequence shown in figure 4-15 is assigned (by the loader) the absolute locations shown in figure 4-16. The LDA 2000,I picks up the *contents* of the location pointed to by ITBL (location 4000_8). The ISZ 2000 indexes the pointer DEF 4000 to point to 4001. The next LDA will reference location 4001, DEF TBL+1. This is what we intend.

label	ABS	m	comments

ABS defines a 16-bit absolute value to be stored at the location represented by the label. The Operand field, m, may be any absolute expression; a single symbol must be defined as absolute elsewhere in the program. Examples are shown in figure 4-17.

```
ITBL  DEF TBL
LDTBL LDA ITBL,I
        .
        .
        .
      ISZ ITBL
        .
        .
        .
TBL   BSS 100
```

Figure 4-15. Example of Correct Address Modification

INSTRUCTION			PAGE	ABSOLUTE LOCATION OF CODE	OPCODE	OPERAND (PAGE) & LOCATION
ITBL	DEF	TBL	(1)	(2000)	DEF	4000
	LDA	ITBL,I	(1)	(2001)	LDA	(1) 2000,I
					.	
					.	
	ISZ	ITBL	(1)	(3000)	ISZ	(1) 2000
					.	
					.	
	TBL	BSS 100	(2)	4000	BSS	

Figure 4-16. Loader-Assigned Locations for Figure 4-10

```
AB    EQU  35        ASSIGNS THE VALUE OF 35
                     TO THE SYMBOL AB
M35   ABS  -AB       M35   CONTAINS -35.
P35   ABS  AB        P35   CONTAINS 35.
P70   ABS  AB+AB     P70   CONTAINS 70.
P30   ABS  AB-5      P30   CONTAINS 30.
P36   ABS  36        P36   CONTAINS 36.
```

Figure 4-17. ABS Examples

label	EQU	m	comments

The EQU pseudo operation assigns to a symbol a value other than the one normally assigned by the program location counter. The symbol in the Label field is assigned the value represnted by the Operand field. The Operand field may contain any expression. The value of the operand may be common, base page or program relocatable as well as absolute,but it should not be negative. Symbols appearing in the operand must be previously defined in the source program.

The EQU instruction may be used to symbolically equate two locations in memory, or it may be used to give a value to a symbol. The EQU statement does not result in a machine instruction. Some examples are shown in figures 4-18 and 4-19.

label	DBL	m	comments

label	DBR	m	comments

Define Left Byte and Define Right Byte (21MX only). The DBL and DBR pseudo instructions each generate one word of memory which contains a 16-bit byte address. For DBL, the byte address being defined is the left half (bits 8-15) of word location *m*; for DBR, it is the right half (bits 0-7). Indirect addressing may *not* be used. A *byte address* is defined as two times the word address of the memory location containing the particular byte. If the byte location is the left half of the memory location (bits 8-15), bit 0 of the byte address is clear; if the byte location is the right half of the memory location (bits 0-7), bit 0 of the byte address is set. In an absolute program, *m* may be any positive expression. In a relocatable program, *m* may be any absolute expression with a value less than 200_8 or any relocatable expression. The generated word may be referenced (via *label*) in the Operand field of LDA and LDB instructions elsewhere in the source program for the purpose of loading byte addresses into the A- and B-registers.

CAUTION

Care must be taken when using the *label* of a DBL or DBR pseudo instruction as an indirect address elsewhere in the source program. The programmer must keep track of whether he is using word addresses or byte addresses.

```
        NAM  FAM
         .
         .
         .
J3      BSS  2          SET ASIDE TWO WORDS FOR STORAGE
         .
         .
         .
        LDA  J3
        ADA  ONE
        STA  J3+1
JFOUR   EQU  J3+1       THE SYMBOLS JFOUR AND J3+1 BOTH IDENTIFY
         .              THE SAME LOCATION. THE "AND" OPERATION
         .              IS PERFORMED ON THIS LOCATION.
         .
MWH     AND  JFOUR
         .
```

Figure 4-18. EQU Example

```
        NAM STOTB
         .
         .
         .
        COM TABLA(10) DEFINES A 10 WORD TABLE, TABLA.
         .
         .
         .
TABLB   EQU TABLA+5    NAMES WORDS 6 THROUGH 10 OF
         .             TABLA AS TABLB.
         .
         .
         .
        LDA TABLB+1    LOADS CONTENTS OF 7TH WORD
         .             COMMON INTO A. THE STATEMENT LDA
         .             TABLA+6 WOULD PERFORM THE SAME
         .             OPERATION
         .
        NAM REG
         .
         .
         .
A       EQU 0          DEFINES SYMBOL A AS 0 (LOCATION
B       EQU 1          OF A-REGISTER), AND SYMBOL B AS
         .             1 (LOCATION OF B-REGISTER).
         .
        LDA B          LOADS CONTENTS OF B-REGISTER
                       INTO A-REGISTER.
```

Figure 4-19. EQU Examples

Examples:

```
BYT1    DBL   WORD1
BYT2    DBR   WORD1
         .
         .
         .
WORD1   NOP
```

If WORD1 has the relocatable address 2002_8, then BYT1 will contain the relocatable value 4004_8 and BYT2 will contain the relocatable value 4005_8.

4-4. CONSTANT DEFINITION

The pseudo instructions in this class enter a string of one or more constant values into consecutive words of the object program. The statements may be named by labels so that other program statements can refer to the fields generated by them.

label	ASC	n, <2n characters>	comments

ASC generates a string of 2n alphanumeric characters in ASCII code into n consecutive words.† One character is right justified in each eight bits; the most significant bit is zero. n may be any expression resulting in an unsigned decimal value in the range 1 through 28. Symbols used in an expression must be previously defined. Anything in the Operand field following 2n characters is treated as comments. If less than 2n characters are detected before the end-of-statement mark, the remaining characters are assumed to be spaces, and are stored as such. The label represents the address of the first two characters. An example is shown in figure 4-20.

label	DEC	$d_1[.d_2,...,d_n]$	comments

DEC records a string of decimal constants into consecutive words. The constants may be either integer or real (floating point), and positive or negative. If no sign is specified, positive is assumed. The decimal number is converted to its

†To enter the code for the ASCII symbols which perform some action (e.g., CR and LF), the OCT pseudo instruction must be used.

```
TTYP   ASC 3,ABCDE
```

causes the following:

NOTE
Columns 7 and 15 below will contain zeros.

ALPHABETIC

	15 14	8 7 6	0
TTYP	A		B
	C		D
	E		(space)

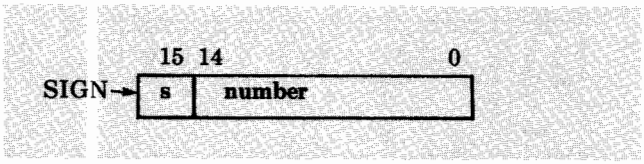
EQUIVALENT IN OCTAL NOTATION

	15 14	8 7 6	0
TTYP	1 0 1		1 0 2
	1 0 3		1 0 4
	1 0 5		0 4 0

Figure 4-20. ASC Example

binary equivalent by the Assembler. The label, if given, serves as the address of the first word occupied by the constant.

A decimal integer must be in the range of -2^{15} to $2^{15}-1$. It is converted into one binary word and appears as follows:

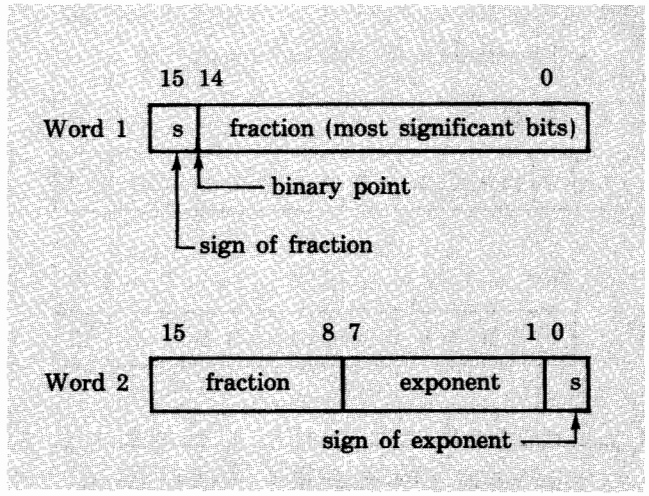


Some examples are shown in figure 4-21.

A floating point number has two components, a fraction and an exponent. The exponent specifies the power of 10 by which the fraction is multiplied. The fraction is a signed or unsigned number which may be written with or without a decimal point. The exponent is indicated by the letter E and follows a signed or unsigned decimal integer. The floating point number may have any of the following formats:

±n.n ±n. ±n.nE±e ±.nE±e ±n.E±e ±nE±e

The number is converted to binary, normalized (leading bits differ), and stored in two computer words. If either the fraction or the exponent is negative, that part is stored in two's complement form.



```
INT    DEC 50,+328,-300,+32768,-32768
```

causes the following (octal representation)

	15	14				0
INT	0	0	0	0	6	2
	0	0	0	5	1	0
	1	7	7	3	2	4
	1	0	0	0	0	0
	1	0	0	0	0	0

Note: The values $\pm 2^{15}$ (± 32768) are both converted to 100000_8.

Figure 4-21. DEC Examples (Integer)

```
DEC .45E1
DEC 45.00E-1
DEC 4500E-3
DEC 4.5
```

are all equivalent to

$.45 \times 10^1$

and are stored in normalized form as:

15	14														0
0	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0

15							8	7						1	0
0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0

Figure 4-22. DEC Examples (Floating Point)

```
DEC -.695,400E-4
```

are stored as:

1	0	1	0	0	1	1	1	0	0	0	0	1	0	1	0

0	0	1	1	1	0	1	1	0	0	0	0	0	0	0	0

0	1	0	1	0	0	0	1	1	1	1	0	1	0	1	1

1	0	0	0	0	1	0	1	1	1	1	1	1	0	0	1

Figure 4-23. DEC Examples (Floating Point)

The floating point number is made up of a 7-bit exponent with sign and a 23-bit fraction with sign. The number must be in the approximate range of 10^{-38} and zero. Examples are shown in figures 4-22 and 4-23.

label	DEX	$d_1[,d_2,...,d_n]$	comments

DEX records a string of extended precision decimal constants into consecutive words within a program. Each such extended precision constant occupies three words as shown in figure 4-24.

An extended precision floating point number is made up of a 39-bit fraction and sign and a 7-bit exponent and sign.

This is the only form used for DEX. All values, whether they be floating point, integer, fraction, or integer and fraction, will be stored in three words as just described. This storage format is basically an extension of that used for DEC, as previously described. Some examples are shown in figure 4-25.

label	OCT	$o_1\ [,o_2,...,o_n]$	comments

OCT stores one or more octal constants in consecutive words of the object program. Each constant consists of one to six octal digits (0 to 177777). If no sign is given, the sign is assumed to be positive. If the sign is negative, the two's complement of the binary equivalent is stored. The constants are separated by commas; the last constant is terminated by a space. If less than six digits are indicated for a constant, the number is right justified in the word. A label, if used, acts as the address of the first constant in the string. The letter B must not be used after the constant in the Operand field; it is significant only when defining an octal term in an instruction other than OCT. Some examples are shown in figure 4-26.

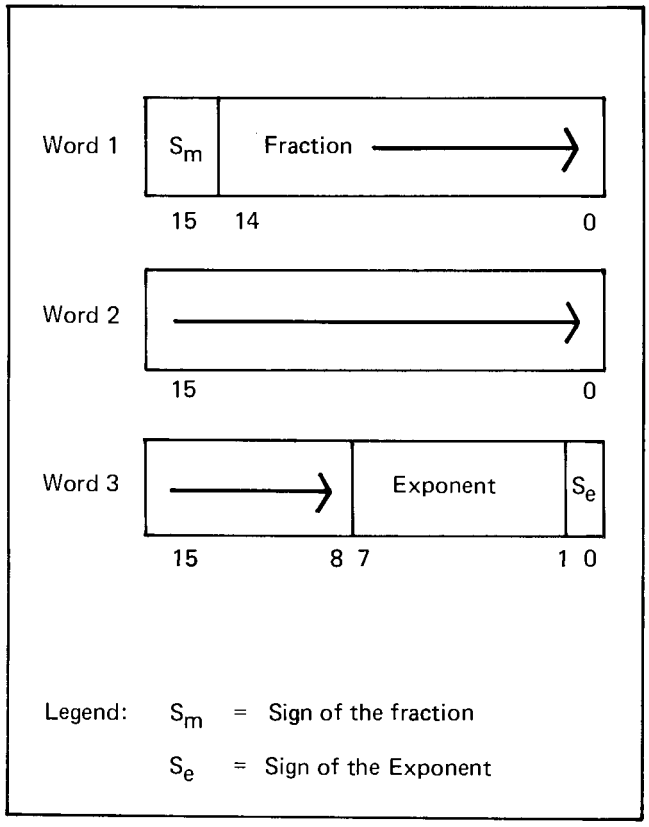


Figure 4-24. DEX Memory Format

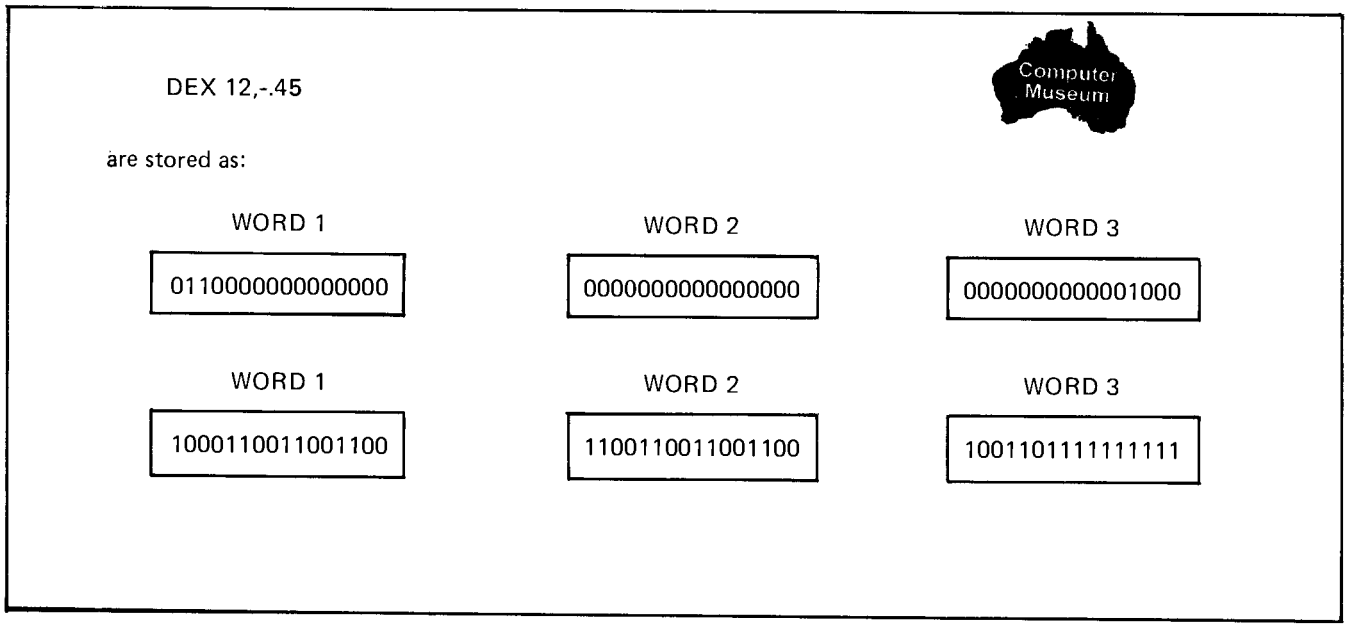


Figure 4-25. DEX Examples

line of coding. These additional code lines are suppressed by an SUP instruction until a UNS or the END pseudo instruction is encountered. SUP will suppress additional code lines in the following machine and pseudo instructions:

ADX	DJS	LAY	MLB	SBY
ADY	DLD	LBX	MPY	SJP
ASC	DST	LBY	MSA	SJS
BYT	FAD	LDX	MSB	STX
CBS	FDV	LDY	MVW	STY
CBT	FMP	MBT	OCT	TBS
CMW	FSB	MCA	SAX	UJP
DEC	JLY	MCB	SAY	UJS
DIV	JPY	MDB	SBS	XMM
DJP	LAX	MLA	SBX	XMS

The SUP pseudo instruction may be used to suppress the listing of literals at the end of the source program listing and also to suppress the printing of offset values for memory reference instructions which refer to external symbols with offsets.

	UNS	comments

The UNS pseudo instruction causes the printing of additional coding lines, terminated by an SUP,to be resumed.

An SUP preceded by another SUP, UNS preceded by UNS, or UNS not preceded by an SUP are not considered errors by the Assembler.

	SKP	comments

The SKP pseudo instruction causes the source program listing to be skipped to the top of the next page. The SKP instruction is not listed, but the source statement sequence number is incremented for the SKP.

	SPC	n

The SPC pseudo instruction causes the source program listing to be skipped a specified number of lines. The list output is skipped n lines, or to the bottom of the page, whichever occurs first. The n may be any absolute expression. The SPC instruction is not listed but the source statement sequence number is incremented for the SPC.

	HED	<heading>

The HED pseudo instruction allows the programmer to specify a heading to be printed at the top of each page of the source program listing.

The heading, a string of up to 56 ASCII characters, is printed at the top of each of the source program listings following the occurrence of the HED pseudo instruction. If HED is the first statement at the beginning of a program, the heading will be used on the first page of the source program listing. A HED instruction placed elsewhere in the program causes a skip to the top of the next page.

The heading specified in the HED pseudo instruction will be used on every page until it is changed by a succeeding HED instruction.

The source statement containing the HED will not be listed, but the source statement sequence number will be incremented.

4-7. ARITHMETIC SUBROUTINE CALLS

If an X appears in the control statement for the source program, the Assembler generates calls to arithmetic subroutines external to the source program for the following instructions: MPY, DIV, DLD, and DST. The instruction formats and functions are as described in paragraph 3-17 of Section III in this manual.

If an F *does not* appear in the control statement for the source program, the Assembler generates calls to arithmetic subroutines external to the source program for the following instructions: FMP, FDV, FAD, and FSB. The instruction formats and functions are as described in paragraph 3-18 of Section III in this manual.

Each use of a statement from this group except FIX and FLT generates two words of instructions. Symbolically, they could be represented as follows:

```
JSB    <.arithmetic pseudo operation>
DEF    m [,I]
```

An EXT <.arithmetic pseudo operation> is implied preceding the JSB operation.

In the above operations, the overflow bit is set when one of the following conditions occurs:

- Integer overflow
- Floating point overflow or underflow
- Division by zero.

Execution of any of the subroutines alter the contents of the E-Register.

4-8. DEFINE USER INSTRUCTION (21MX SERIES ONLY)

	MIC	opcode,fcode,pnum	comments

This pseudo instruction provides the user the capability of defining his own instructions. *opcode* is a three-character alphabetic mnemonic, *fcode* is an instruction code, and *pnum* declares how many (0-7) parameter addresses are to be associated with the newly-defined instruction. Both *fcode* and *pnum* may be expressions which generate an absolute result. A user-defined instruction must not appear in the source program prior to the MIC pseudo instruction which defines it. When the user-defined mnemonic is used later in the source program, the specified number of parameter addresses (*pnum*) are supplied in the Operand field of the user-defined instruction separated from one another by spaces. The parameter addresses may be any addressable values, relocatable and/or indirect. The parameters may not be literals.

Note: All three operands (*opcode, fcode,* and *pnum*) must be supplied in the MIC pseudo instruction in order for the specified instruction to be defined. If *pnum* is zero, it must be expressly declared as such (*not* omitted).

4-9. "JUMP TO MICROPROGRAM"

The MIC pseudo instruction is primarily intended to facilitate the passing of control from an assembly language program to a user's microprogram residing in Read-Only-Memory (ROM) or Writable Control Store (WCS). Ordinarily, to do this the user must include an OCT 101*xxx* or OCT 105*xxx* statement (where *xxx* is 140 through 737) at the point in the source program where the jump is to occur. If parameters are to be passed, they are usually defined as constants (via OCT or DEF statements) immediately following the OCT 105*xxx* statement. With the MIC pseudo instruction, the user can define a source language instruction which passes control and a series of parameter addresses to a microprogram. If it is desired to pass additional parameters to a microprogram beyond those pointed to by the user-defined instruction, they must be defined as constants (via OCT or DEF statements) immediately following each use of the user-defined instruction.

4-10. EXAMPLE. Assume that the first two parameters to be passed from the assembly language program to the user's microprogram reside in the memory locations PARM1 and PARM2 and that the third parameter resides in the memory location pointed to by ADR. Also assume that the octal code for transferring control to the particular microprogram is 105240_8.

The following statement defines a source language instruction which passes control and three parameter addresses to the microprogram:

```
MIC    ABC,105240B,3
```

Whenever it is desired to pass control from the assembly language program to the microprogram, the following user-defined instruction may be used in the source program:

```
ABC    PARM1 PARM2 ADR,I
```

4-11. COMBINING MULTIPLE MNEMONICS

Another use of the MIC pseudo instruction is to assign a single mnemonic to a multiple instruction (shift-rotate or alter-skip) statement.

4-12. EXAMPLE. Instead of using the source statement:

```
ALR,CLE,SLA,RAL
```

the user may define a single mnemonic as follows:

```
MIC    XYZ,01472B,0
```

where 01472B is the octal instruction code for the four-mnemonic statement shown above. Whenever XYZ is subsequently used as an instruction mnemonic in the source program, it is the equivalent of using the source statement:

```
ALR,CLE,SLA,RAL
```

4-13. DEFINING CONSTANTS

The MIC pseudo instruction may also be used for defining constants (*opcode* = mnemonic, *fcode* = constant, and

pnum = 0). Whenever the defined mnemonic is used as an instruction mnemonic in the source program the Assembler automatically replaces it with the specified constant.

4-14. EXAMPLE. The following statement defines the constant 10_{10} and assigns it the mnemonic TEN:

```
MIC   TEN,10,0
```

Whenever TEN appears as an instruction mnemonic later in the source program, the value 10_{10} is automatically inserted in that location by the Assembler.

HP CHARACTER SET FOR COMPUTER SYSTEMS

APPENDIX A

Effect of Control key *

000-037B | 040-077B | 100-137B | 140-177B

b_7 b_6 b_5					0_0 0_0	0_0 0_1	0_1 0_0	0_1 0_1	1_0 0_0	1_0 0_1	1_1 0_0	1_1 0_1
BITS				COLUMN →	0	1	2	3	4	5	6	7
b_4	b_3	b_2	b_1	ROW ↓								
0	0	0	0	0	NUL	DLE	SP	0	@	P		p
0	0	0	1	1	SOH	DC1	!	1	A	Q	a	q
0	0	1	0	2	STX	DC2	"	2	B	R	b	r
0	0	1	1	3	ETX	DC3	#	3	C	S	c	s
0	1	0	0	4	EOT	DC4	$	4	D	T	d	t
0	1	0	1	5	ENQ	NAK	%	5	E	U	e	u
0	1	1	0	6	ACK	SYN	&	6	F	V	f	v
0	1	1	1	7	BEL	ETB	'	7	G	W	g	w
1	0	0	0	8	BS	CAN	(	8	H	X	h	x
1	0	0	1	9	HT	EM	)	9	I	Y	i	y
1	0	1	0	10	LF	SUB	*	:	J	Z	j	z
1	0	1	1	11	VT	ESC	+	;	K	[	k	{
1	1	0	0	12	FF	FS	,	<	L	\	l	¦
1	1	0	1	13	CR	GS	–	=	M	]	m	}
1	1	1	0	14	SO	RS	.	>	N	^	n	~
1	1	1	1	15	SI	US	/	?	O	_	o	DEL

32 CONTROL CODES

Upshifted Lower Case

64 CHARACTER SET

96 CHARACTER SET

128 CHARACTER SET

EXAMPLE: The representation for the character "K" (column 4, row 11) is.

	b_7	b_6	b_5	b_4	b_3	b_2	b_1
BINARY	1	0	0	1	0	1	1
OCTAL	1	1			3		

* Depressing the Control key while typing an upper case letter produces the corresponding control code on most terminals. For example, Control-H is a backspace.

9206- 1A

HEWLETT-PACKARD CHARACTER SET FOR COMPUTER SYSTEMS

This table shows HP's implementation of ANS X3.4-1968 (USASCII) and ANS X3.32-1973. Some devices may substitute alternate characters from those shown in this chart (for example, Line Drawing Set or Scandanavian font). Consult the manual for your device.

The left and right byte columns show the octal patterns in a 16 bit word when the character occupies bits 8 to 14 (left byte) or 0 to 6 (right byte) and the rest of the bits are zero. To find the pattern of two characters in the same word, add the two values. For example, "AB" produces the octal pattern 040502. (The parity bits are zero in this chart.)

The octal values 0 through 37 and 177 are control codes. The octal values 40 through 176 are character codes.

Decimal Value	Octal Values		Mnemonic	Graphic[1]	Meaning
	Left Byte	Right Byte			
0	000000	000000	NUL	NU	Null
1	000400	000001	SOH	SH	Start of Heading
2	001000	000002	STX	SX	Start of Text
3	001400	000003	ETX	EX	End of Text
4	002000	000004	EOT	ET	End of Transmission
5	002400	000005	ENQ	EQ	Enquiry
6	003000	000006	ACK	AK	Acknowledge
7	003400	000007	BEL	🔔	Bell, Attention Signal
8	004000	000010	BS	BS	Backspace
9	004400	000011	HT	HT	Horizontal Tabulation
10	005000	000012	LF	LF	Line Feed
11	005400	000013	VT	VT	Vertical Tabulation
12	006000	000014	FF	FF	Form Feed
13	006400	000015	CR	CR	Carriage Return
14	007000	000016	SO	SO	Shift Out } Alternate
15	007400	000017	SI	SI	Shift In } Character Set
16	010000	000020	DLE	DL	Data Link Escape
17	010400	000021	DC1	D1	Device Control 1 (X-ON)
18	011000	000022	DC2	D2	Device Control 2 (TAPE)
19	011400	000023	DC3	D3	Device Control 3 (X-OFF)
20	012000	000024	DC4	D4	Device Control 4 ($\overline{\text{TAPE}}$)
21	012400	000025	NAK	NK	Negative Acknowledge
22	013000	000026	SYN	SY	Synchronous Idle
23	013400	000027	ETB	EB	End of Transmission Block
24	014000	000030	CAN	CN	Cancel
25	014400	000031	EM	EM	End of Medium
26	015000	000032	SUB	SB	Substitute
27	015400	000033	ESC	EC	Escape[2]
28	016000	000034	FS	FS	File Separator
29	016400	000035	GS	GS	Group Separator
30	017000	000036	RS	RS	Record Separator
31	017400	000037	US	US	Unit Separator
127	077400	000177	DEL	▩	Delete, Rubout[3]

Decimal Value	Octal Values		Character	Meaning
	Left Byte	Right Byte		
32	020000	000040		Space, Blank
33	020400	000041	!	Exclamation Point
34	021000	000042	"	Quotation Mark
35	021400	000043	#	Number Sign, Pound Sign
36	022000	000044	$	Dollar Sign
37	022400	000045	%	Percent
38	023000	000046	&	Ampersand, And Sign
39	023400	000047	'	Apostrophe, Acute Accent
40	024000	000050	(	Left (opening) Parenthesis
41	024400	000051	)	Right (closing) Parenthesis
42	025000	000052	*	Asterisk, Star
43	025400	000053	+	Plus
44	026000	000054	,	Comma, Cedilla
45	026400	000055	–	Hyphen, Minus, Dash
46	027000	000056	.	Period, Decimal Point
47	027400	000057	/	Slash, Slant
48	030000	000060	0	} Digits, Numbers
49	030400	000061	1	
50	031000	000062	2	
51	031400	000063	3	
52	032000	000064	4	
53	032400	000065	5	
54	033000	000066	6	
55	033400	000067	7	
56	034000	000070	8	
57	034400	000071	9	
58	035000	000072	:	Colon
59	035400	000073	;	Semicolon
60	036000	000074	<	Less Than
61	036400	000075	=	Equals
62	037000	000076	>	Greater Than
63	037400	000077	?	Question Mark

Decimal Value	Octal Values Left Byte	Octal Values Right Byte	Character	Meaning
64	040000	000100	@	Commercial At
65	040400	000101	A	Upper Case Alphabet, Capital Letters
66	041000	000102	B	
67	041400	000103	C	
68	042000	000104	D	
69	042400	000105	E	
70	043000	000106	F	
71	043400	000107	G	
72	044000	000110	H	
73	044400	000111	I	
74	045000	000112	J	
75	045400	000113	K	
76	046000	000114	L	
77	046400	000115	M	
78	047000	000116	N	
79	047400	000117	O	
80	050000	000120	P	
81	050400	000121	Q	
82	051000	000122	R	
83	051400	000123	S	
84	052000	000124	T	
85	052400	000125	U	
86	053000	000126	V	
87	053400	000127	W	
88	054000	000130	X	
89	054400	000131	Y	
90	055000	000132	Z	
91	055400	000133	[	Left (opening) Bracket
92	056000	000134	\	Backslash, Reverse Slant
93	056400	000135	]	Right (closing) Bracket
94	057000	000136	^ ↑	Caret, Circumflex, Up Arrow[4]
95	057400	000137	_ ←	Underline, Back Arrow[4]

Decimal Value	Octal Values Left Byte	Octal Values Right Byte	Character	Meaning
96	060000	000140	`	Grave Accent[5]
97	060400	000141	a	Lower Case Letters[5]
98	061000	000142	b	
99	061400	000143	c	
100	062000	000144	d	
101	062400	000145	e	
102	063000	000146	f	
103	063400	000147	g	
104	064000	000150	h	
105	064400	000151	i	
106	065000	000152	j	
107	065400	000153	k	
108	066000	000154	l	
109	066400	000155	m	
110	067000	000156	n	
111	067400	000157	o	
112	070000	000160	p	
113	070400	000161	q	
114	071000	000162	r	
115	071400	000163	s	
116	072000	000164	t	
117	072400	000165	u	
118	073000	000166	v	
119	073400	000167	w	
120	074000	000170	x	
121	074400	000171	y	
122	075000	000172	z	
123	075400	000173	{	Left (opening) Brace[5]
124	076000	000174	|	Vertical Line[5]
125	076400	000175	}	Right (closing) Brace[5]
126	077000	000176	~	Tilde, Overline[5]

Notes: [1]This is the standard display representation. The software and hardware in your system determine if the control code is displayed, executed, or ignored. Some devices display all control codes as "·", "@", or space.

[2]Escape is the first character of a special control sequence. For example, ESC followed by "J" clears the display on a 2640 terminal.

[3]Delete may be displayed as "__", "@", or space.

[4]Normally, the caret and underline are displayed. Some devices substitute the up arrow and back arrow.

[5]Some devices upshift lower case letters and symbols (` through ~) to the corresponding upper case character (@ through ^). For example, the left brace would be converted to a left bracket.

HP 7970B BCD-ASCII CONVERSION

SYMBOL	BCD (OCTAL CODE)	ASCII EQUIVALENT (OCTAL CODE)	SYMBOL	BCD (OCTAL CODE)	ASCII EQUIVALENT (OCTAL CODE)
(space)	20	040	@	14	100
!	52	041	A	61	101
"	37	042	B	62	102
#	13	043	C	63	103
$	53	044	D	64	104
%	57	045	E	65	105
&	†	046	F	66	106
'	35	047	G	67	107
(	34	050	H	70	110
)	74	051	I	71	111
*	54	052	J	41	112
+	60	053	K	42	113
,	33	054	L	43	114
-	40	055	M	44	115
.	73	056	N	45	116
/	21	057	O	46	117
0	12	060	P	47	120
1	01	061	Q	50	121
2	02	062	R	51	122
3	03	063	S	22	123
4	04	064	T	23	124
5	05	065	U	24	125
6	06	066	V	25	126
7	07	067	W	26	127
8	10	070	X	27	130
9	11	071	Y	30	131
:	15	072	Z	31	132
;	56	073	[	75	133
<	76	074	\	36	134
=	17	075	]	55	135
>	16	076	↑	77	136
?	72	077	←	32	137

†The ASCII code 046 is converted to the BCD code for a space (20) when writing data onto a 7-track tape.

SUMMARY OF INSTRUCTIONS

APPENDIX B

Symbols	Meaning
label	Symbolic label, 1-5 alphanumeric characters and periods
m	Memory location represented by an expression
I	Indirect addressing indicator
C	Clear flag indicator
(m,m+1)	Two-word floating point value in m and m+1
comments	Optional comments
[ ]	Optional portion of field
{ }	One of set may be selected
P	Program Counter
( )	Contents of location
$\wedge$	Logical product
$\not\vee$	Exclusive "or"
$\vee$	Inclusive "or"
A	A-register
B	B-register
E	E-register
A_n	Bit n of A-register
B_n	Bit n of B-register
b	Bit positions in B- and A-register
$(\overline{A/B})$	Complement of contents of register A or B
(AB)	Two-word floating point value in register A and B
sc	Channel select code represented by an expression
d	Decimal constant
o	Octal constant
r	Repeat count
n	Integer constant
lit	Literal value
msb	Most significant bits
lsb	Least significant bits

B-1. MACHINE INSTRUCTIONS

B-2 MEMORY REFERENCE

B-3. Jump and Increment-Skip

ISZ	m [,I]	(m) + 1 → m: then if (m) = 0, execute P + 2 otherwise execute P + 1
JMP	m [,I]	Jump to m; m → P
JSB	m [,I]	Jump subroutine to m: P + 1 → m; m + 1 → P

B-4. Add, Load and Store

ADA	$\left\{\begin{matrix} \text{m [,I]} \\ \text{lit} \end{matrix}\right\}$	(m) + (A) → A
ADB	$\left\{\begin{matrix} \text{m [,I]} \\ \text{lit} \end{matrix}\right\}$	(m) + (B) → B
LDA	$\left\{\begin{matrix} \text{m [,I]} \\ \text{lit} \end{matrix}\right\}$	(m) → A
LDB	$\left\{\begin{matrix} \text{m [,I]} \\ \text{lit} \end{matrix}\right\}$	(m) → B
STA	m [,I]	(A) → m
STB	m [,I]	(B) → m

B-5. Logical

AND	$\left\{\begin{matrix} \text{m [,I]} \\ \text{lit} \end{matrix}\right\}$	(m) ∧ (A) → A
XOR	$\left\{\begin{matrix} \text{m [,I]} \\ \text{lit} \end{matrix}\right\}$	(m) ∀ (A) → A
IOR	$\left\{\begin{matrix} \text{m [,I]} \\ \text{lit} \end{matrix}\right\}$	(m) ∨ (A) → A
CPA	$\left\{\begin{matrix} \text{m [,I]} \\ \text{lit} \end{matrix}\right\}$	If (m) ≠ (A), execute P + 2, otherwise execute P + 1
CPB	$\left\{\begin{matrix} \text{m [,I]} \\ \text{lit} \end{matrix}\right\}$	If (m) ≠ (B), execute P + 2, otherwise execute P + 1

B-6. Word Processing

MVW	$\left\{\begin{matrix} \text{m [,I]} \\ \text{lit} \end{matrix}\right\}$	Move (m) words from array (A) → array (B)
CMW	$\left\{\begin{matrix} \text{m [,I]} \\ \text{lit} \end{matrix}\right\}$	Compare (m) words of array (A) against (m) words of array (B); if the two arrays are equal, execute P + 3, if array (A) is less than array (B), execute P + 4, if array (A) is greater than array (B), execute P + 5

B-7. Byte Processing

LBT		B contains a 16-bit byte address; $((B)) \rightarrow A_{0-7}$; o's to A_{8-15}
SBT		B contains a 16-bit byte address; $(A_{0-7}) \rightarrow (B)$
MBT	$\left\{ \begin{matrix} m\ [,I] \\ lit \end{matrix} \right\}$	A and B contain 16-bit byte addresses; move (m) bytes from array (A)→array (B)
CBT	$\left\{ \begin{matrix} m\ [,I] \\ lit \end{matrix} \right\}$	A and B contain 16-bit byte addresses; compare (m) bytes of array (A) against (m) bytes of array (B); if the two arrays are equal, execute P + 3; if array (A) is less than array (B), execute P + 4; if array (A) is greater than array (B), execute P + 5
SFB		A_{0-7} contain the test byte, A_{8-15} contain the termination byte, and B contains a 16-bit byte address; scan array (B); if test byte found, execute P + 1, B contains address of test byte; if termination byte found, execute P + 2, B contains address of termination byte; if neither is found, execute P + 2, B contains zero

B-8. Bit Processing

TBS	$\left\{ \begin{matrix} m\ [,I] \\ lit \end{matrix} \right\}$	n [,I]	Compare all "set" bits in (m) against corresponding bits in (n); if all bits tested are set, execute P + 3; if any of the bits tested are clear, execute P + 4
SBS	$\left\{ \begin{matrix} m\ [,I] \\ lit \end{matrix} \right\}$	n [,I]	Set all bits in (n) which correspond to "set" bits in (m)
CBS	$\left\{ \begin{matrix} m\ [,I] \\ lit \end{matrix} \right\}$	n [,I]	Clear all bits in (n) which correspond to "set" bits in (m)

B-9. REGISTER REFERENCE

B-10. Shift-Rotate

CLE	$0 \rightarrow E$
ALS	Shift (A) left one bit, $0 \rightarrow A_0$, A_{15} unaltered
BLS	Shift (B) left one bit, $0 \rightarrow B_0$, B_{15} unaltered
ARS	Shift (A) right one bit, $(A_{15}) \rightarrow A_{14}$
BRS	Shift (B) right one bit, $(B_{15}) \rightarrow B_{14}$
RAL	Rotate (A) left one bit
RBL	Rotate (B) left one bit
RAR	Rotate (A) right one bit
RBR	Rotate (B) right one bit
ALR	Shift (A) left one bit, $0 \rightarrow A_{15}$
BLR	Shift (B) left one bit, $0 \rightarrow B_{15}$
ERA	Rotate E and A right one bit
ERB	Rotate E and B right one bit
ELA	Rotate E and A left one bit
ELB	Rotate E and B left one bit
ALF	Rotate A left four bits
BLF	Rotate B left four bits
SLA	If $(A_0) = 0$, execute P + 2, otherwise execute P + 1
SLB	If $(B_0) = 0$, execute P + 2, otherwise execute P + 1

Shift-Rotate instructions can be combined as follows:

$$\left[\begin{Bmatrix}\text{ALS}\\\text{ARS}\\\text{RAL}\\\text{RAR}\\\text{ALR}\\\text{ALF}\\\text{ERA}\\\text{ELA}\end{Bmatrix}\right] \quad [,\text{CLE}] \quad [,\text{SLA}] \quad \left[,\begin{Bmatrix}\text{ALS}\\\text{ARS}\\\text{RAL}\\\text{RAR}\\\text{ALR}\\\text{ALF}\\\text{ERA}\\\text{ELA}\end{Bmatrix}\right]$$

$$\left[\begin{Bmatrix}\text{BLS}\\\text{BRS}\\\text{RBL}\\\text{RBR}\\\text{BLR}\\\text{BLF}\\\text{ERB}\\\text{ELB}\end{Bmatrix}\right] \quad [,\text{CLE}] \quad [,\text{SLB}] \quad \left[,\begin{Bmatrix}\text{BLS}\\\text{BRS}\\\text{RBL}\\\text{RBR}\\\text{BLR}\\\text{BLF}\\\text{ERB}\\\text{ELB}\end{Bmatrix}\right]$$

B-11. No-Operation

NOP	Execute P + 1

B-12. Alter-Skip

CLA	0's → A
CLB	0's → B
CMA	$\overline{(A)} \rightarrow A$
CMB	$\overline{(B)} \rightarrow B$
CCA	1's → A
CCB	1's → B
CLE	0 → E
CME	$\overline{(E)} \rightarrow E$
CCE	1 → E
SEZ	If (E) = 0, execute P + 2, otherwise execute P + 1
SSA	If $(A_{15}) = 0$, execute P + 2, otherwise execute P + 1
SSB	If $(B_{15}) = 0$, execute P + 2, otherwise execute P + 1
INA	(A) + 1 → A
INB	(B) + 1 → B
SZA	If (A) = 0, execute P + 2, otherwise execute P + 1
SZB	If (B) = 0, execute P + 2, otherwise execute P + 1
SLA	If $(A_0) = 0$, execute P + 2, otherwise execute P + 1
SLB	If $(B_0) = 0$, execute P + 2, otherwise execute P + 1
RSS	Reverse sense of skip instructions. If no skip instructions precede, execute P + 2

Alter-Skip instructions can be combined as follows:

$$\left[\left\{\begin{matrix}\text{CLA}\\\text{CMA}\\\text{CCA}\end{matrix}\right\}\right] \quad [,\text{SEZ}] \quad \left[,\left\{\begin{matrix}\text{CLE}\\\text{CME}\\\text{CCE}\end{matrix}\right\}\right] \quad [,\text{SSA}]\ [,\text{SLA}]\ [,\text{INA}]\ [,\text{SZA}]\ [,\text{RRS}]$$

$$\left[\left\{\begin{matrix}\text{CLB}\\\text{CMB}\\\text{CCB}\end{matrix}\right\}\right] \quad [,\text{SEZ}] \quad \left[,\left\{\begin{matrix}\text{CLE}\\\text{CME}\\\text{CCE}\end{matrix}\right\}\right] \quad [,\text{SSB}]\ [,\text{SLB}]\ [,\text{INB}]\ [,\text{SZB}]\ [,\text{RSS}]$$

B-13. Index Register

CAX		(A)→X
CBX		(B)→X
CAY		(A)→Y
CBY		(B)→Y
CXA		(X)→A
CXB		(X)→B
CYA		(Y)→A
CYB		(Y)→B
XAX		(A)→X and (X)→A
XBX		(B)→X and (X)→B
XAY		(A)→Y and (Y)→A
XBY		(B)→Y and (Y)→B
ISX		(X) + 1→X, then test new (X); if (X) = 0, execute P + 2, otherwise execute P + 1
ISY		(Y) + 1→Y, then test new (Y); if (Y) = 0, execute P + 2, otherwise execute P + 1
DSX		(X) - 1→X, then test new (X); if (X) = 0, execute P + 2, otherwise execute P + 1
DSY		(Y) - 1→Y, then test new (Y); if (Y) = 0, execute P + 2, otherwise execute P + 1
LDX	m [,I] lit	(m)→X
LDY	m [,I] lit	(m)→Y
STX	m [,I]	(X)→m
STY	m [,I]	(Y)→m
LAX	m [,I]	(m + (X))→A
LBX	m [,I]	(m + (X))→B
LAY	m [,I]	(m + (Y))→A
LBY	m [,I]	(m + (Y))→B
SAX	m [,I]	(A)→m + (X)
SBX	m [,I]	(B)→m + (X)
SAY	m [,I]	(A)→m + (Y)
SBY	m [,I]	(B)→m + (Y)
ADX	m [,I] lit	(m) + (X)→X
ADY	m [,I] lit	(m) + (Y)→Y
JLY	m [,I]	Jump to m; P + 2→Y
JPY	m	Jump to m + (Y)

Computer Museum

B-14. INPUT/OUTPUT, OVERFLOW, AND HALT

B-15. Input/Output

STC	sc [,C]	Set control bit_{sc}, enable transfer of one element of data between $device_{sc}$ and $buffer_{sc}$
CLC	sc [,C]	Clear control bit_{sc}. If sc = 0 clear all control bits
LIA	sc [,C]	$(buffer_{sc}) \rightarrow A$
LIB	sc [,C]	$(buffer_{sc})$ V $(A) \rightarrow A$ Merge (inclusive or) the buffer into A
MIA	sc [,C]	$(buffer_{sc})$ V $(B) \rightarrow B$ Merge (inclusive or) the buffer into B.
MIB	sc [,C]	$(buffer_{sc})$ $(B) \rightarrow B$
OTA	sc [,C]	$(A) \rightarrow buffer_{sc}$
OTB	sc [,C]	$(B) \rightarrow buffer_{sc}$
STF	sc	Set flag bit_{sc}. If sc = 0, enable interrupt system. sc = 1 sets overflow bit.
CLF	sc	Clear flag bit_{sc}. If sc = 0, disable interrupt system. If sc = 1, clear overflow bit.
SFC	sc	If (flag bit_{sc}) = 0, execute P + 2, otherwise execute P + 1. If sc = 1, test overflow bit.
SFS	sc	If (flag bit_{sc}) = 1, execute P + 2, otherwise execute P + 1. If sc = 1, test overflow bit.

B-16. Overflow

CLO		$0 \rightarrow$ overflow bit
STO		$1 \rightarrow$ overflow bit
SOC	[C]	If (overflow bit) = 0, execute P + 2, otherwise execute P + 1
SOS	[C]	If (overflow bit) = 0, execute P + 2, otherwise execute P + 1

B-17. Halt

HLT	[sc [,C]]	Halt computer

B-18. EXTENDED ARITHMETIC UNIT

MPY	$\left\{\begin{matrix} m\ [,I] \\ lit \end{matrix}\right\}$	$(A) \times (m) \rightarrow (B_{\pm msb}$ and $A_{lsb})$
DIV	$\left\{\begin{matrix} m\ [,I] \\ lit \end{matrix}\right\}$	$(B_{\pm msb}$ and $A_{lsb})/(m) \rightarrow A$, remainder $\rightarrow B$
DLD	$\left\{\begin{matrix} m\ [,I] \\ lit \end{matrix}\right\}$	(m) and $(m + 1) \rightarrow A$ and B
DST	$\left\{\begin{matrix} m\ [,I] \\ lit \end{matrix}\right\}$	(A) and $(B) \rightarrow m$ and m + 1
ASR	b	Arithmetically shift (BA) right b bits, B_{15} extended
ASL	b	Arithmetically shift (BA) left b bits, B_{15} unaltered, 0's to A_{lsb}

RRR	b	Rotate (BA) right b bits
RRL	b	Rotate (BA) left b bits
LSR	b	Logically shift (BA) right b bits, 0's to B_{msb}
LSL	b	Logically shift (BA) left b bits, o's to A_{lsb}
SWP		Swap the contents of the A and B registers

B-19. FLOATING POINT

FMP	$\left\{ \begin{matrix} m\ [,I] \\ lit \end{matrix} \right\}$	(AB) x (m, m + 1)→AB
FDV	$\left\{ \begin{matrix} m\ [,I] \\ lit \end{matrix} \right\}$	(AB)/(m, m + 1)→AB
FAD	$\left\{ \begin{matrix} m\ [,I] \\ lit \end{matrix} \right\}$	(m, m + 1) + (AB)→AB
FSB	$\left\{ \begin{matrix} m\ [,I] \\ lit \end{matrix} \right\}$	(AB) - (m, m + 1)→AB
FIX		(AB) converted from floating-point to fixed-point; result→A
FLT		(A) converted from fixed-point to floating-point; result→AB

B-20. MEMORY EXPANSION

DJP	m [,I]	Disable MEM and jump to m; m → P
DJS	m [,I]	Disable MEM and jump subroutine to m; P + 1 → m; m + 1 → P
JRS	m_1 [,I] m_2 [,I]	Jump and restore status
LFA		A → fence
LFB		B → fence
MBF		Move bytes from alternate map. X← 0; A← A + no. bytes moved; B← B + no. bytes moved.
MBI		Move bytes into alternate map. X← 0; A← A + no. bytes moved; B← B + no. bytes moved.
MBW		Move bytes within alternate map. X← 0; A← A + no. bytes moved; B← B + no. bytes moved.
MWF		Move words from alternate map. X← 0; A← A + no. words moved; B← B + no. words moved.
MWI		Move words into alternate map. X← 0; A← A + no. words moved; B← B + no. words moved.
MWW		Move words within alternate map. X← 0; A← A + no. words moved; B← B + no. words moved.

PAA		If A(15) = 0, Port A map ← memory; if A(15) = 1, Port A map → memory.
PAB		If B(15) = 0, Port A map ← memory; if B(15) = 1, Port A map → memory.
PBA		If A(15) = 0, Port B map ← memory; if A(15) = 1, Port B map → memory.
PBB		If B(15) = 0, Port B map ← memory; if B(15) = 1, Port B map → memory.
RSA		A ← status register
RSB		B ← status register
RVA		A ← violation register
RVB		B ← violation register
SJP	m [,I]	Enable System map and jump to m
SJS	m [,I]	Enable System map and jump subroutine to m
SSM	m [,I]	m ← status register
SYA		If A(15) = 0, System map ← memory; if A(15) = 1, System map → memory.
SYB		If B(15) = 0, System map ← memory; if B(15) = 1, System map → memory.
UJP	m [,I]	Enable User map and jump to m
UJS	m [,I]	Enable User map and jump subroutine to m
USA		If A(15) = 0, User map ← memory; if A(15) = 1, User map → memory.
USB		If B(15) = 0, User map ← memory; if B(15) = 1, User map → memory.
XCA	m [,I]	Compare A with m; if A = m, execute P = 1; if A ≠ m, execute P + 2.
XCB	m [,I]	Compare B with m; if B = m, execute P + 1; if B ≠ m, execute P + 2.
XLA	m [,I]	A ← m
XLB	m [,I]	B ← m
XMA		If A(15) = 0 and A(0) = 0, Port A map ← System map. If A(15) = 0 and A(0) = 1, Port B map ← system map. If A(15) = 1 and A(0) = 0, Port A map ← User map. If A(15) = 1 and A(0) = 1, Port B map ← User map.
XMB		If B(15) = 0 and B(0) = 0, Port A map ← System map. If B(15) = 0 and B(0) = 1, Port B map ← System map. If B(15) = = 1 and B(0) = 0, Port A map ← User map. If B(15) = 1 and B(0) = 1, Port B map ← User map.
XMM		A = register no., B = memory address, X = no. of registers. If X > 0, Maps ← memory; if X < 0, Memory ← maps.
XMS		A = first register no., B = first page no., X = positive no. of registers. First register is loaded with the page number indicated in B, the second register is loaded with that value + 1, and so forth.
XSA	m [,I]	A → m
XSB	m [,I]	B → m

B-21. PSEUDO INSTRUCTIONS

B-22. ASSEMBLER CONTROL

NAM	[name]	Specifies relocatable program and its name.
ORB		Gives relocatable program origin for the base page of relocatable program.
ORG	m	Gives absolute program origin or origin for a segment of relocatable or absolute program.
ORR		Reset main program location counter at value existing when first ORG or ORB of a string was encountered.
END	[m]	Terminates source language program. Produces transfer to program starting location, m, if given.
REP <statement>	r	Repeat immediately following statement r times.
IFN <statements> XIF		Include statements in program if control statement contains N.
IFZ <statements> XIF		Include statements in program if control statement contains Z.

B-23. OBJECT PROGRAM LINKAGE

COM $name_1[(size_1)][,name_2[(size_2)],...,name_n[(size_n)]]$

Reserves a block of common storage locations. $name_1$ identifies segments of block, each of length size.

ENT $name_1[,name_2,...,name_n]$

Defines entry points, $name_1$, that may be referred to by other programs.

EXT $name_1[,name_2, . . .,name_n]$

Defines external locations, name , which are labels of other programs, referenced by this program.

label RPL [m]

Defines the code replacement for [JSB label] external references.

B-24. ADDRESS AND SYMBOL DEFINITION

label	DEF m [,I]	Generates a 15-bit address which may be referenced indirectly through the label.
label	ABS m	Defines a 16-bit absolute value to be referenced by the label.

label	EQU m	Equates the value, m, to the label.
label	DBL m	Defines a 16-bit byte address (left half, bits 8-15, of word location *m*) to be referenced by the label.
label	DBR m	Defines a 16-bit byte address to be referenced by the label. The byte address is for the right half (bits 0-7) of word location m.

B-25. CONSTANT DEFINITION

ASC n, <2n characters>	Generates a string of 2n ASCII characters.
DEC d_1 [,d_2,...,d_n]	Records a string of decimal constants of the form: Integer: ±n Floating point: ±n.n, ±n., ±.n, ±nE±e, ±n.nE±e, ±n.E±e, ±.nE±e
DEX d_1 [,d_2,...,d_n]	Records a string of extended precision decimals constants of the form Floating point: ±n, ±n.m, ±n., ±.n, ±nE±e, ±n.nE±e, ±n.E±e, ±.nE±e
OCT o_1 [,o_2,...,o_n]	Records a string of octal constants of the form: ±000000
BYT b [,b ,...,b_n]	Records a string of octal byte constants of the form: ±nnn (where *nnn* is 0 through 377_8).

B-26. STORAGE ALLOCATION

BSS	m	Reserves a storage area of length, m.

B-27. ASSEMBLY LISTING CONTROL

UNL		Suppress assembly listing output.
LST		Resume assembly listing output.
SKP		Skip listing to top of next page.
SPC	n	Skip n lines on listing.
SUP		Suppress listing of extended code lines (e.g., as produced by subroutine calls).
UNS		Resume listing of extended code lines.
HED	<heading>	Print <heading> at top of each page, where <heading> is up to 56 ASCII characters.

B-28. DEFINE USER INSTRUCTION

MIC	opcode,fcode,pnum	Defines a source language instruction. *opcode* = three-character alphabetic mnemonic, *fcode* = instruction code, and *pnum* declares how many parameter addresses are to be associated with the newly-defined instruction.

ALPHABETIC LIST OF INSTRUCTIONS

APPENDIX C

Note: In the following list, those instructions suffixed with an asterisk are dynamic mapping instructions and cannot be used unless the computer contains a Dynamic Mapping System.

ABS	Define absolute value
ADA	Add to A
ADB	Add to B
ADX	Add memory to X
ADY	Add memory to Y
ALF	Rotate A left 4
ALR	Shift A left 1, clear sign
ALS	Shift A left 1
AND	"And" to A
ARS	Shift A right 1, sign carry
ASC	Generate ASCII characters
ASL	Arithmetic long shift left
ASR	Arithmetic long shift right
BLF	Rotate B left 4
BLR	Shift B left 1, clear sign
BLS	Shift B left 1
BRS	Shift B right 1, carry sign
BSS	Reserve block of storage starting at symbol
BYT	Defines octal byte constants
CAX	Copy A to X
CAY	Copy A to Y
CBS	Clear bits
CBT	Compare bytes
CBX	Copy B to X
CBY	Copy B to Y
CCA	Clear and complement A (1's)
CCB	Clear and complement B (1's)
CCE	Clear and complement E (set E = 1)
CLA	Clear A
CLB	Clear B
CLC	Clear I/O control bit
CLE	Clear E
CLF	Clear I/O flag
CLO	Clear overflow bit
CMA	Complement A
CMB	Complement B
CME	Complement E
CMW	Compare words
COM	Reserve block of common storage
CPA	Compare to A, skip if unequal
CPB	Compare to B, skip if unequal
CXA	Copy X to A
CXB	Copy X to B
CYA	Copy Y to A
CYB	Copy Y to B
DBL	Define left byte (bits 8-15) address
DBR	Define right byte (bits 0-7) address
DEC	Define decimal constant
DEF	Define address
DEX	Define extended precision constant
DIV	Divide
DJP*	Disable MEM and jump
DJS*	Disable MEM and jump to subroutine
DLD	Double load
DST	Double store
DSX	Decrement X and skip if zero
DSY	Decrement Y and skip if zero
ELA	Rotate E and A left 1
ELB	Rotate E and B left 1
END	Terminate program
ENT	Entry point
ERA	Rotate E and A right 1
ERB	Rotate E and B right 1
EQU	Equate symbol
EXT	External reference
FAD	Floating add
FDV	Floating divide
FIX	Convert floating-point be fixed-point
FLT	Convert fixed-point to floating-point
FMP	Floating multiply
FSB	Floating subtract

HED	Print heading at top of each page
HLT	Halt
IFN	When N appears in Control statement, assemble ensuing instructions
IFZ	When Z appears in Control statement, assemble ensuing instructions
INA	Increment A by 1
INB	Increment B by 1
IOR	Inclusive "or" to A
ISX	Increment X and skip if zero
ISY	Increment Y and skip if zero
ISZ	Increment, then skip if zero
JLY	Jump and load Y
JMP	Jump
JPY	Jump indexed by Y
JRS*	Jump and restore status
JSB	Jump to subroutine
LAX	Load A from memory indexed by X
LAY	Load A from memory indexed by Y
LBT	Load byte
LBX	Load B from memory indexed by X
LBY	Load B from memory indexed by Y
LDA	Load into A
LDB	Load into B
LDX	Load X from memory
LDY	Load Y from memory
LFA*	Load fence from A
LFB*	Load fence from B
LIA	Load into A from I/O channel
LIB	Load into B from I/O channel
LSL	Logical long shift left
LSR	Logical long shift right
LST	Resume list output (follows a UNL)
MBF*	Move bytes from alternate map
MBI*	Move bytes into alternate map
MBT	Move bytes
MBW*	Move bytes within alternate map
MIA	Merge (or) into A from I/O channel
MIB	Merge (or) into B from I/O channel
MIC	Define jump to user microcode
MPY	Multiply
MVW	Move words
MWF*	Move words from alternate map
MWI*	Move words into alternate map
MWW*	Move words within alternate map
NAM	Name relocatable program
NOP	No operation
OCT	Define octal constant
ORB	Establish origin in base page
ORG	Establish program origin
ORR	Reset program location counter
OTA	Output from A to I/O channel
OTB	Output from B to I/O channel
PAA*	Load/store Port A map per A
PAB*	Load/store Port A map per B
PBA*	Load/store Port B map per A
PBB*	Load/store Port B map per B
RAL	Rotate A left 1
RAR	Rotate A right 1
RBL	Rotate B left 1
RBR	Rotate B right 1
REP	Repeat next statement
RPL	Replace instruction definition
RRL	Rotate A and B left
RRR	Rotate A and B right
RSA*	Read status register into A
RSB*	Read status register into B
RSS	Reverse skip sense
RVA*	Read violation register into A
RVB*	Read violation register into B
SAX	Store A into memory indexed by X
SAY	Store A into memory indexed by Y
SBS	Set bits
SBT	Store byte
SBX	Store B into memory indexed by X
SBY	Store B into memory indexed by Y
SEZ	Skip if E = 0
SFB	Scan for byte
SFC	Skip if I/O flag = 0 (clear)
SFS	Skip if I/O flag = 1 (set)

SJP*	Enable System map and jump
SJS*	Enable System map and jump to subroutine
SKP	Skip to top of next page
SLA	Skip if LSB of A = 0
SLB	Skip if LSB of B = 0
SOC	Skip if overflow bit = 0 (clear)
SOS	Skip if overflow bit = 1 (set)
SPC	Space n lines
SSA	Skip if sign A = 0
SSB	Skip if sign B = 0
SSM*	Store status register in memory
STA	Store A
STB	Store B
STC	Set I/O control bit
STF	Set I/O flag
STO	Set overflow bit
STX	Store X into memory
STY	Store Y into memory
SUP	Suppress list output of additional code lines
SWP	Switch A and B
SYA*	Load/store System map per A
SYB*	Load/store System map per B
SZA	Skip if A = 0
SZB	Skip if B = 0
TBS	Test bits
UJP*	Enable User map and jump
UJS*	Enable User map and jump to subroutine
UNL	Suppress list output
UNS	Resume list output
USA*	Load/store User map per A
USB*	Load/store User map per B
XAX	Exchange A and X
XAY	Exchange A and Y
XBX	Exchange B and X
XBY	Exchange B and Y
XCA*	Cross compare A
XCB*	Cross compare B
XIF	Terminate IFN or IFZ group of instructions
XLA*	Cross load A
XLB*	Cross load B
XMA*	Transfer maps internally per A
XMB*	Transfer maps internally per B
XMM*	Transfer map or memory
XMS*	Transfer maps sequentially
XOR	Exclusive "or" to A
XSA*	Cross store A
XSB*	Cross store B

CONSOLIDATED CODING SHEETS

APPENDIX D

Table D-1 presents the binary codes for the base set instructions while Table D-2 presents those for the extended instruction group.

Computer Museum

Table D-1. Base Set Instruction Codes in Binary

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
D/I	AND		001	0	Z/C	← Memory Address →									
D/I	XOR		010	0	Z/C										
D/I	IOR		011	0	Z/C										
D/I	JSB		001	1	Z/C										
D/I	JMP		010	1	Z/C										
D/I	ISZ		011	1	Z/C										
D/I	AD*		100	A/B	Z/C										
D/I	CP*		101	A/B	Z/C										
D/I	LD*		110	A/B	Z/C										
D/I	ST*		111	A/B	Z/C										
15	**14**	**13**	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
0	SRG		000	A/B	0	D/E	*LS	000		†CLE	D/E	‡SL*	*LS	000	
				A/B	0	D/E	*RS	001			D/E		*RS	001	
				A/B	0	D/E	R*L	010			D/E		R*L	010	
				A/B	0	D/E	R*R	011			D/E		R*R	011	
				A/B	0	D/E	*LR	100			D/E		*LR	100	
				A/B	0	D/E	ER*	101			D/E		ER*	101	
				A/B	0	D/E	EL*	110			D/E		EL*	110	
				A/B	0	D/E	*LF	111			D/E		*LF	111	
				NOP	000			000			000			000	
15	**14**	**13**	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
0	ASG		000	A/B	1	CL*	01	CLE	01	SEZ	SS*	SL*	IN*	SZ*	RSS
				A/B		CM*	10	CME	10						
				A/B		CC*	11	CCE	11						
15	**14**	**13**	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
1	IOG		000		1	H/C	HLT	000		← Select Code →					
					1	0	STF	001							
					1	1	CLF	001							
					1	0	SFC	010							
					1	0	SFS	011							
				A/B	1	H/C	MI*	100							
				A/B	1	H/C	LI*	101							
				A/B	1	H/C	OT*	110							
				0	1	H/C	STC	111							
				1	1	H/C	CLC	111							
					1	0	STO	001			000			001	
					1	1	CLO	001			000			001	
					1	H/C	SOC	010			000			001	
					1	H/C	SOS	011			000			001	
15	**14**	**13**	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
1	EAG		000	MPY**		000	010				000			000	
				DIV**		000	100				000			000	
				DLD**		100	010				000			000	
				DST**		100	100				000			000	
				ASR		001	000			0	1				
				ASL		000	000			0	1				
				LSR		001	000			1	0		number		
				LSL		000	000			1	0		← of →		
				RRR		001	001			0	0		bits		
				RRL		000	001			0	0				

Notes: * = A or B, according to bit 11.
D/I, A/B, Z/C, D/E, H/C coded: 0/1.
**Second word is Memory Address.

†CLE: Only this bit is required.
‡SL*: Only this bit and bit 11 (A/B as applicable) are required.

Table D-2. Extended Instruction Group Codes in Binary

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
SAX/SAY/SBX/SBY	1	0	0	0	A/B	0	1	1	1	1	1	0	X/Y	0	0	0
CAX/CAY/CBX/CBY	1	0	0	0	A/B	0	1	1	1	1	1	0	X/Y	0	0	1
LAX/LAY/LBX/LBY	1	0	0	0	A/B	0	1	1	1	1	1	0	X/Y	0	1	0
STX/STY	1	0	0	0	1	0	1	1	1	1	1	0	X/Y	0	1	1
CXA/CYA/CXB/CYB	1	0	0	0	A/B	0	1	1	1	1	1	0	X/Y	1	0	0
LDX/LDY	1	0	0	0	1	0	1	1	1	1	1	0	X/Y	1	0	1
ADX/ADY	1	0	0	0	1	0	1	1	1	1	1	0	X/Y	1	1	0
XAX/XAY/XBX/XBY	1	0	0	0	A/B	0	1	1	1	1	1	0	X/Y	1	1	1
ISX/ISY/DSX/DSY	1	0	0	0	1	0	1	1	1	1	1	1	X/Y	0	0	I/D
JUMP INSTRUCTIONS	1	0	0	0	1	0	1	1	1	1	1	1		0	1	0
													JLY = 0 JPY = 1			
BYTE INSTRUCTIONS	1	0	0	0	1	0	1	1	1	1	1	1	0			
													LBT = 0 1 1 SBT = 1 0 0 MBT = 1 0 1 CBT = 1 1 0 SFB = 1 1 1			
BIT INSTRUCTIONS	1	0	0	0	1	0	1	1	1	1	1	1	1			
													SBS = 0 1 1 CBS = 1 0 0 TBS = 1 0 1			
WORD INSTRUCTIONS	1	0	0	0	1	0	1	1	1	1	1	1	1	1	1	
														CMW = 0 MVW = 1		

Table D-2. Extended Instruction Group Codes in Binary (Continued)

MEMORY EXPANSION

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

DJP = 0 1 0
DJS = 0 1 1

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
SYB/USB/PAB/PBB/SSM/JRS	1	0	0	0	1	0	1	1	1	1	0	0	1			

SYB = 0 0 0
USB = 0 0 1
PAB = 0 1 0
PBB = 0 1 1
SSM = 1 0 0
JRS = 1 0 1

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
XMA/XLA/XSA/XCA/LFA	1	0	0	0	0	0	1	1	1	1	0	1	0			

XMA = 0 1 0
XLA = 1 0 0
XSA = 1 0 1
XCA = 1 1 0
LFA = 1 1 1

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
MBI/MBF/MBW/MWI/MWF/MWW	1	0	0	0	1	0	1	1	1	1	0	0	0			

MBI = 0 1 0
MBF = 0 1 1
MBW = 1 0 0
MWI = 1 0 1
MWF = 1 1 0
MWW = 1 1 1

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
SYA/USA/PAA/PBA	1	0	0	0	0	0	1	1	1	1	0	0	1			

SYA = 0 0 0
USA = 0 0 1
PAA = 0 1 0
PBA = 0 1 1

Table D-2. Extended Instruction Group Codes in Binary (Continued)

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
XMM/XMS/XMB/XLB XSB/XCB/LFB	1	0	0	0	1	0	1	1	1	1	0	1	0			

XMM = 0 0 0
XMS = 0 0 1
XMB = 0 1 0
XLB = 1 0 0
XSB = 1 0 1
XCB = 1 1 0
LFB = 1 1 1

RSA/RVA	1	0	0	0	0	0	1	1	1	1	0	1	1			

RSA = 0 0 0
RVA = 0 0 1

RSB/RVB/SJP/SJS/UJP/UJS	1	0	0	0	1	0	1	1	1	1	0	1	1			

RSB = 0 0 0
RVB = 0 0 1
SJP = 1 0 0
SJS = 1 0 1
UJP = 1 1 0
UJS = 1 1 1

RUNNING ASSEMBLIES

APPENDIX E

The Assembler, a segmented program that executes in the main-memory User Program Area, operates under control of RTE. The Assembler consists of a main program (ASMB) and five segments (ASMBD, ASMB1, ASMB2, ASMB3, ASMB4), and resides on the disc. The main program is read into main memory when called by a ON directive.

Source pograms, accepted from either an input device or a user source file on the disc, are translated into absolute or relocatable object programs; absolute code is punched in binary records, suitable for execution only outside of RTE. ASMB can store relocatable code in the LG Area of the disc for on-line execution, as well as punch it on paper tape.

A source program passes through the input device only once, unless there is insufficient disc storage space. In the latter case, the Assembler informs the user that two passes are required.

E-1. ASSEMBLER I/O

The Assembly Language I/O EXEC calls should specify the proper logical unit numbers for the system configuration.

If the memory protect hardware option is present (and enabled), it protects the resident supervisor from alteration. It interrupts the execution of a user program under these conditions:

- Any operation that would modify the protected area or jump into it.
- Any I/O instruction, except those referencing the switch register or overflow register.
- The halt instruction.

Memory protect gives control to RTE when an interrupt occurs, and RTE checks whether it was an EXEC call. If not, the user program is aborted.

E-2. ASSEMBLER OPERATION

The RTE Assembler is initiated with a ON directive. However, before entering the ON directive, the operator must place the source program in the input device. If the source program is on the disc, the operator must first set up the LS Area with the editor, and set parameter $p_1 = 2$ in the ON directive. The ON directive for Assembler should take the following form:

```
:ON,ASMB[,P1,P2,P3,P4,99]
```

where:

P_1
logical unit number of input device (default is 5; set to 2 for source file input indicated by a LS directive or set by the editor)

P_2
Logical unit number of list device (default is the lu of the interactive input device)

P_3
logical unit number of punch device (default is 4)

P_4
lines/page on the source listing (default is 56)

99
the LG parameter. If present, the object program is stored in the LG Area for later relocation. Any requested punch output still occurs. (The *99* may occur anywhere in the parameter list, but terminates the list.)

All parameters are optional. Parameters P_1 through P_4, however, are positional (if present they must appear in the order shown above). Thus, unless all of the parameters P_1 through P_4 are omitted, the associated (trailing) comma must be included to denote which parameters are omitted.

For example,

ON,ASMB,,P2,,P4	(P_1 and P_3 omitted)
ON,ASMB,P1,,P3,P4	(P_2 omitted)

If the *99* is omitted, the binary output is not placed in the LG Area.

E-3. MESSAGES DURING ASSEMBLY

When the end of a source tape is encountered, the following is output on the system console:

```
I/O ET L #x E #y S #z
```

LU #x is unavailable until the operator declares the associated EQT up/using the RTE UP command:

```
UP,y
```

Compilation continues after the *UP*. More than one source tape can be compiled into one program by loading the next tape before giving the *UP*.

The following message on the system console signifies the end of assembly:

```
$END ASMB
```

If another pass of the source program is required, this message is output at the end of pass one.

```
$END ASMB PASS
```

The operator must replace the program in the input device and enter:

```
GO,ASMB
```

If an error is found in the Assembler control statement, the following message is output on the system console:

```
$END ASMB CS
```

and the current assembly stops.

If an end-of-file condition on the source input occurs before an END statement is found, the console signals:

```
$END ASMB XEND
```

and the current assembly stops.

If source input from logical unit 2 (disc) is requested, but no file has been declared, the system console signals:

```
$END ASMB NPRG
```

and the current assembly stops.

If the *99* parameter is edited, but no LG tracks were allocated, the following message is printed on the system console:

```
IO06
```

The Assembler is then aborted.

If the LG Area, where binary code is stored by a *99* parameter, overflows, assembly continues but the following message is output on the system console:

```
IO09
```

The Assembler is then aborted.

The next message is printed on a separate line just above each error diagnostic printed in the program listing during pass 1.

```
# nnn
```

nnn is the "tape" number on which the error (reported on the next line of the listing) occurred. A program may consist of more than one tape. The tape counter starts with one and increments by one whenever an end-of-tape condition occurs (paper tape) or a blank card is encountered. When the counter increments, the numbering of source statements starts over at one.

Each error diagnostic printed in the program listing during pass 2 of the assembly is associated with a different message (printed on a separate line just above each diagnostic):

```
PG ppp
```

ppp is the page number (in the listing) of the *previous* error diagnostic. PG 000 is associated with the first error found in the program.

THE FORMATTER

APPENDIX F

NOTE

This appendix is included for those users who wish to run programs under the Basic Control System (BCS)

The formatter is a subroutine that can be called by relocatable programs to perform formatted data transfers, to interpret formats, to provide unformatted input and output binary data, to provide free-field input, and to provide buffer-to-buffer conversion. The formatter is first given a string of ASCII characters that constitutes a format code. This code tells the formatter the variables to transfer, the order and the conversion (on input, ASCII characters are converted to binary values and on output, binary values are converted to ASCII characters). Then the calling program gives the formatter a string of variables to be output or filled by input.

F-1. INPUT AND OUTPUT

In languages such as FORTRAN and ALGOL, when the programmer uses a READ or WRITE statement the compiler generates all the necessary calls to the formatter.

In assembly language, however, the programmer is responsible for all calls to the formatter. For each I/O operation, the programmer must first make an initialization call (to entry point .DIO. for decimal input/output, or .BIO. for binary input/output). This call establishes the format to be used (if any), the logical unit, and whether the operation is input or output. Then, for each data item, the program must make a separate call which depends on the type of data. Finally, for output only, the program must make a termination call that tells the formatter to output the last record.

The flowchart in figure F-1 shows the process of selecting an input calling sequence and figure F-2 contains a flowchart showing the output calling sequence.

Variable items in the calling sequence include:

unit is the logical unit number of the desired I/O device.

format is the label of an assembly language ASC pseudo-instruction that defines the format specification.

end of list is the location following the last data call to the formatter. When an error occurs in the format specification or the input data, the formatter returns to this location.

real is the address of the real variable.

integer is the address of the integer variable.

length is the number of elements (not the number of memory locations) in the array. Maximum length of an external physical record is 67 words for formatted data and 60 words for binary data. Formatted data blocks can be of any length if the format breaks the data in multiple records using "/" and unlimited groups. If binary data exceeds 60 words, the record is read in or written out and the formatter skips to the next record.

(Note: For this reason, binary data should be read in with the same variable list as that used to write it out.)

address is the first location of the array.

F-2. RECORDS

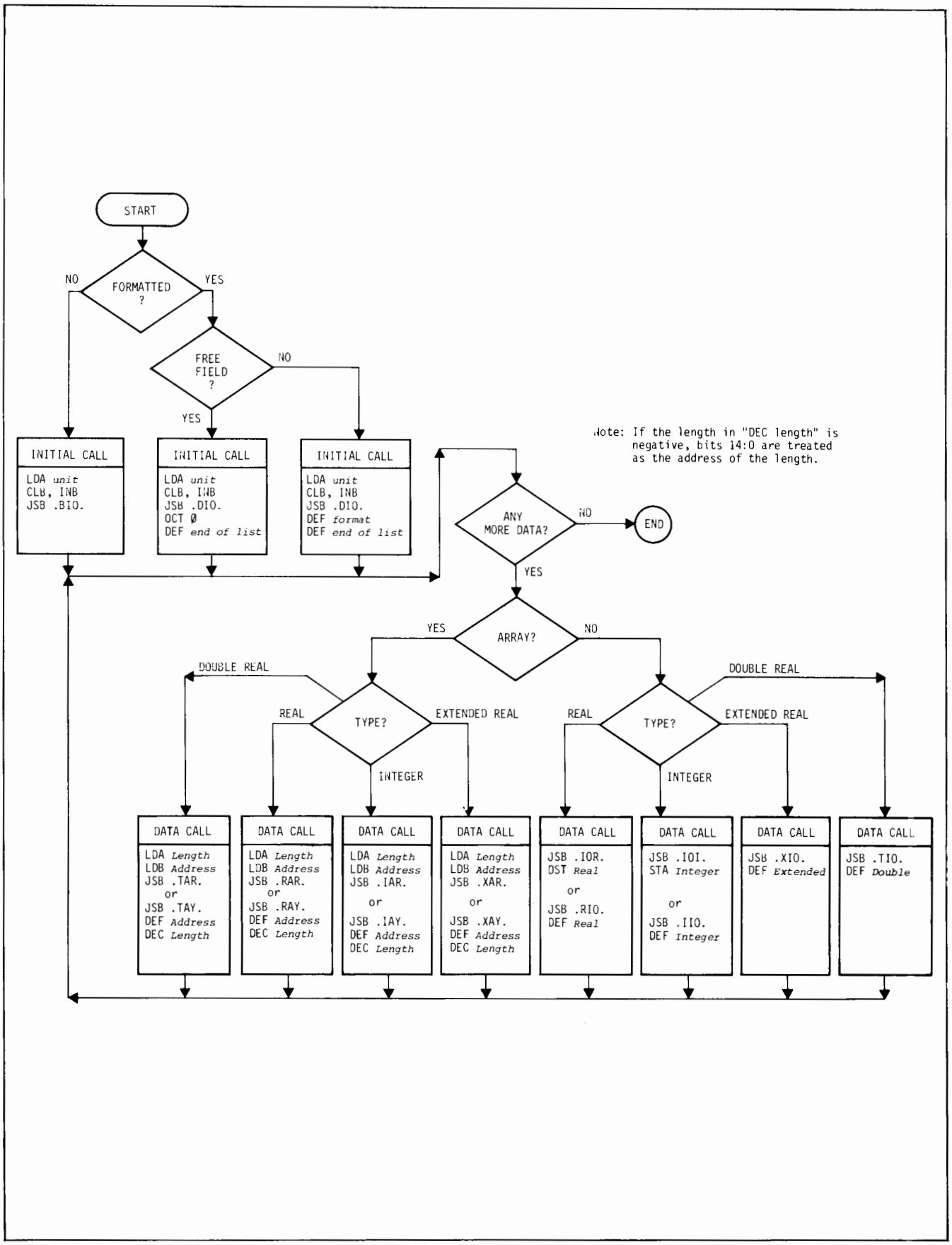
Unformatted I/O through the formatter is limited to physical records of 60 words maximum. If a variable list contains more than 60 words of data, the data is broken into more than one record. (For example, 100 words of data are broken into two records of 60 and 40 words each.)

When paper tape or unit record devices are used (teleprinter, mark sense card reader, etc.), however, only 59 words of each record contain data. The first word issued is for the record length.

F-3. FORMATTED INPUT/OUTPUT

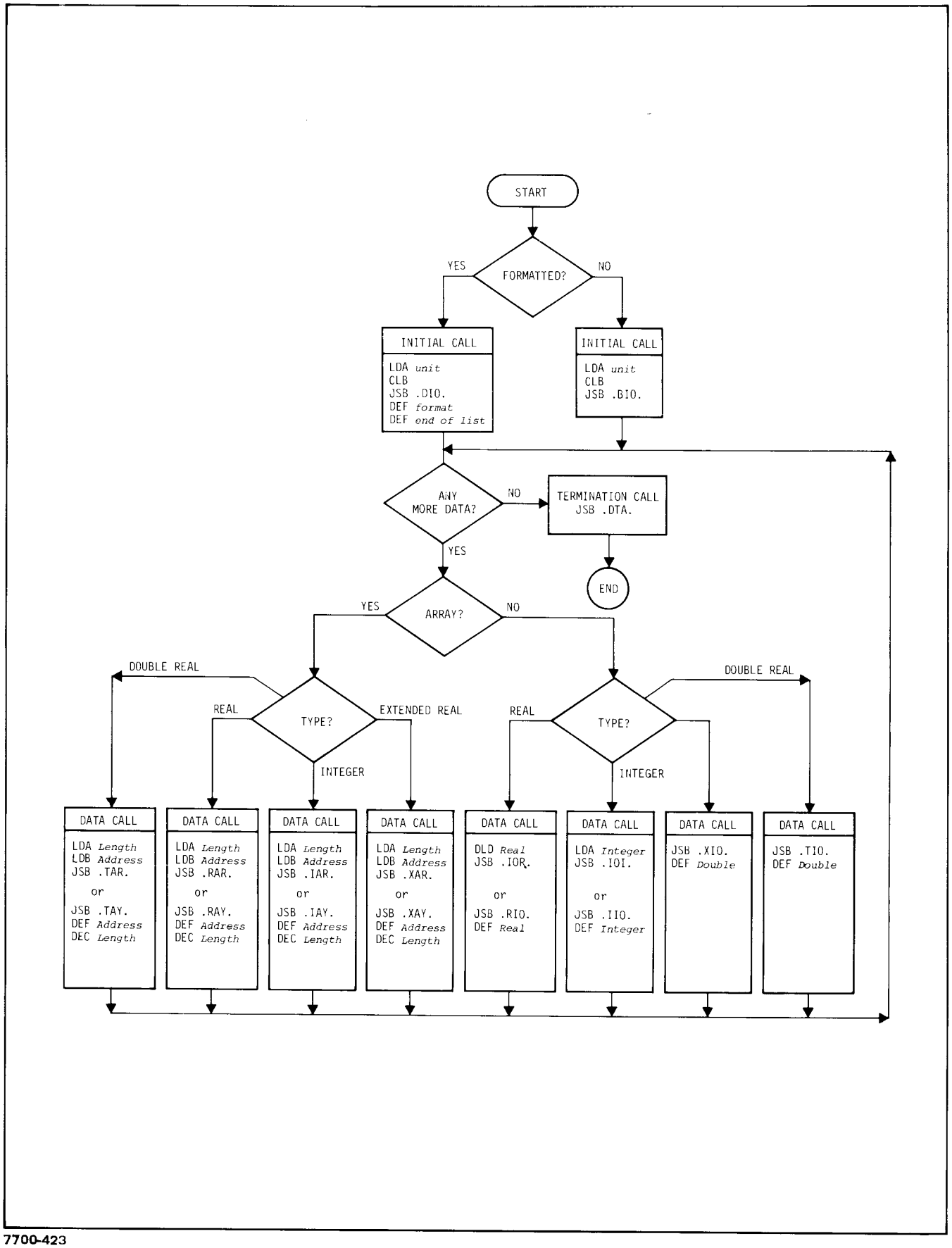
Formatted input/output is distinguished from unformatted input/output by the presence of an ASCII string format specification. The ASCII characters consist of a series of format specifications or codes. Each code specifies either a conversion or an editing operation. Conversion specifications tell the formatter how to handle each variable in the data list.

Format specifications may be nested (enclosed in parentheses) to a depth of one level. Conversion specifications tell the formatter how to convert variables into ASCII output and how to convert ASCII input into binary variable data. Editing specifications tell the formatter what literal strings to output, when to begin new records, and when to insert blanks.



7700-422

Figure F-1. Input Calling Sequence Selection



7700-423

Figure F-2. Output Calling Sequence Selection

F-4. FORMAT SPECIFICATIONS

A format has the following form:

(*spec*, . . . ,*r*(*spec*,. . .),*spec*, . .)

where:

spec
is a format specification.

r
is an optional repeat factor which must be an integer.

F-5. CONVERSION SPECIFICATIONS

Conversion specifications are as follows:

*r*E*w.d*	Real number with exponent (E specification)		
*r*F*w.d*	Real number without exponent (F specification)		
*r*I*w*	Decimal integer (I specification)		
r@*w*	Octal integer	O, K, and @ specifications	Not available with 4k formatter
*r*K*w*, *r*O*w*	Octal integer		
*r*A*w*, *r*R*w*	ASCII character	(A and R specifications)	

where:

r
is an integer repetition factor.

w
is a non-zero integer constant representing the width of the field.

d
is an integer constant representing the digital fraction.

F-6. EDITING SPECIFICATIONS

Editing specifications are as follows:

*n*X	Blank field
*n*H	Character string
r"	Character string
r/	*Begin new record.*

where:

n
is a non-zero integer constant representing the width of the field in the external character string.

r
is an integer repetition factor.

F-7. E SPECIFICATION

The E specification defines a field for a real number with exponent.

F-8. OUTPUT. On output, the E specification converts numbers (integer or real) in memory into character form. The E field is defined in a format by the presence of the E specification (Ew.d). The field is w positions in the output record. The variable is printed in the field as

$$\overbrace{\underbrace{-.x_1 \ldots x_d}_{d}\ E\pm ee}^{w}$$

where:

$x_1 \ldots x_d$
are the most significant digits of the value.

ee
are the digits of the exponent

w
is the width of the field.

d
is the number of significant digits.

–
is present if the number is negative.

The w must be large enough to contain the significant digits (d), the sign, the decimal point, E, and the exponent. In general, w should be greater than or equal to d + 6.

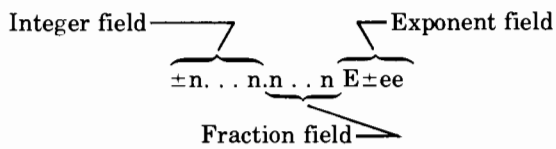
If w is greater than the number of positions required for the output value, the quantity is right justified in the field with spaces to the left. If w is not large enough (e.g., less than d + 6), then the value of d is truncated to fit in the field. If this is not possible, the entire field is filled with dollar signs ($).

Examples

FORMAT	DATA ITEM	RESULT
E10.3	+12.34	bb.123E+02
E10.3	−12.34	b−.123E+02
E12.4	−12.34	bbb.1234E+02
E12.4	−12.34	bb−.1234E+02
E7.3	+12.34	.12E+02
E5.1	+12.34	$$$$$

F-9. INPUT. On input, the E specification tells the formatter to interpret the next w positions in the record as a real number with exponent. The formatter then converts the field into a number and stores it into the variable specified in the variable list.

The input field may consist of integer, fraction, and exponent subfields:



where the format equals Ew.d.

F-10. Rules for E Field Input. Rules for E field input are as follows:

1. The width of the input item must not be greater than w characters, with w > d.

2. Initial + and E are optional, for example,

   123. = +123.

   12.+6 = 12.E6

3. If E is present, the initial + of the exponent is optional, for example,

   123.4E06

4. If the decimal point is left out, the formatter inserts it by multiplying the integer field by 10^{-d}, for example,

   if format = E9.4, then

   123456E+6 = 12.3456E+6

5. Any combination of integer field, fraction field, and exponent field is legal:

   123.456E6

   .456E6

   .456

   123.E6

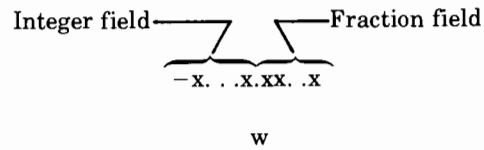
   123.

   E6

F-11. F SPECIFICATION

The F specification defines a field for a fixed-point real number (no exponent).

F-12. OUTPUT. On output, the F specification converts numbers (integer or real) in a format by the presence of the F specification (Fw.d). The field is w positions in the output record. The variable is printed out right-justified in fixed-point form with d digits to the right of the decimal point:



where:

w
is the total width of the field.

d
is the length of the fraction field (if d = 0, fraction field is empty)

−
is present if the number is a negative value (positive numbers can be unsigned).

If w is greater than the number of positions required for the output value, the quantity is right-justified in the field with spaces to the left. If w is not large enough to hold the data item, then the value of d is reduced to fit. If this is not possible, the entire field is filled with dollar signs ($).

Examples:

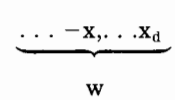
FORMAT	DATA ITEM	RESULT
F10.3	+12.34	bbbb12.340
F10.3	−12.34	bbb−12.340
F12.3	+12.34	bbbbbb12.340
F12.3	−12.34	bbbbb−12.340
F4.3	+12.34	12.3
F4.3	+12345.12	$$$$

F-13. INPUT. Input to an F field is identical to an E field. All the rules under the E specification apply equally to the F specification.

F-14. I SPECIFICATION

The I specification defines a field for a decimal integer.

F-15. OUTPUT. On output, the I specification converts numbers (integer or real) in memory into character form. The I field is defined in a format by the presence of the I specification (Iw). The field occupies w positions in the output record. The variable is converted to an integer, if necessary, and printed out right-justified in the field (spaces to the left) as:



where:

$x_1 \ldots x_d$
are the digits of the value (maximum = 5).

w
is the width of the field in characters.

–
is present if the number is negative.

If the output field is too short, the entire field is filled with dollar signs ($).

Examples:

FORMAT	DATA ITEM	RESULT
I5	−1234	−1234
I5	+12345	12345
I4	+12345	$$$$
I6	+12345	b12345

F-16. INPUT. The I specification on input is equivalent to the Fw.0 specification. The input field is read in, the number is converted to the form suitable to the variable (integer or real), and the binary value is stored in the variable location.

During input, if a value is less than -32768_{10}, it is converted to +32767.

Examples:

FORMAT	INPUT FIELD	INTERNAL RESULT
I5	−b123	−123
I5	12003	12003
I4	b102	102
I1	3	3

F-17. O, K, AND @ SPECIFICATIONS

The O, K, and @ specification types are equivalent; they all are used to convert octal (base eight) numbers.

F-18. OUTPUT. On output, the octal specifications (O, K, @) convert an integer value in memory into octal digits for output. The octal field is defined by the presence of the O (Ow), K (Kw), or @ (@w) specification. The field is w octal digits wide. The integer value is converted and right-justified in the field as:

$$\underbrace{\ldots bbd_1 \ldots d_n}_{w}$$

where:

$d_1 \ldots d_n$
are the octal digits (6 maximum).

. . .bb
are lead spaces.

w
is the width of the field.

If w is less than 6, the w least significant digits are written.

F-19. INPUT. On input, the octal specification tells the formatter to interpret the next w positions in the input record as an octal number. The formatter converts the digits into an octal integer and stores this value in an integer variable.

If w is greater than or equal to six, up to six octal digits are stored. Non-octal digits with the field are ignored.

If w is less than six or if less than six octal digits occur in the field, the result is right-justified in the variable with zeros to the left. If the value of the octal digits in the field is greater than 177777, the results are unpredictable.

Examples:

FORMAT	INPUT FIELD	INTERNAL RESULT
@6	123456	123456
@7	−123456	123456
2K5	2342342342	023423 and 042342
2@4	,396E−05	000036 and 000005

F-20. A AND R SPECIFICATIONS

The A and R specifications define a field of one or two ASCII characters. ASCII characters are stored as two 8-bit codes per integer variable, four 8-bit codes per real variable, six per extended real, and eight per double real. The number of characters per variable will always be referred to as "v".

F-21. OUTPUT. On output, the A and R specifications transfer ASCII character codes from memory to an external medium. The field is defined by an A or R specification (Aw or Rw). The field is w positions wide in the output record. For w ≥ v, A and R are equivalent: the field is blank filled to the left of the data. For w>v, the A specification uses the left-most characters in the variable, and the R specification (and A if OLDIO) uses the right-most.

Examples:

VARIABLE	FORMAT	OUTPUT FORMAT
ABCD	A4 & R4	ABCD
ABCD	A6 & R6	bbABCD
ABCD	A3	ABC
ABCD	R2	CD

A string of n*v characters may be output from (or input to) n variables (e.g. using an array of length n) using a repeat factor.

Examples:

VARIABLE TYPE	VARIABLES	FORMAT	INPUT OR OUTPUT
4 integers	AB,CD,EF,GH	4A2	ABCDEFGH
2 reals	ABCD,EFGH	2A4	ABCDEFGH
1 double real	ABCDEFGH	A8	ABCDEFGH

F-22. INPUT. On input, the A and R specifications transfer ASCII character codes from an external medium to internal memory. The field is defined by an A or R specification (Aw or Rw) and is w positions wide. If w > v, the rightmost characters are taken from the input field. For the A specification with w<v, data is left-justified and blank filled in the variable. For the R specification (and A if OLDIO) with w<v, data is right justified and zero-filled.

Examples:

INPUT FIELD	FORMAT	REAL VARIABLE
MN	A2	MNbb
MN	R2	zzMN
MNOP	A4,R4	MNOP
MNOPQRS	A7,R7	PQRS

z=binary zero

F-23. X SPECIFICATION

The X specification produces spaces on output and skips characters on input. The comma following X in the format is optional.

F-24. OUTPUT. On output, the X specification causes spaces to be inserted in the output record. The X field is defined by the presence of an X specification (nX) in the format, where n is the number of spaces to be inserted. (X alone = 1X; 0X is not permitted.)

Examples:

FORMAT

E8.3,5X,F6.2,5X,I4

DATA VALUES

+123.4, −12.34, −123

OUTPUT FIELD

.123E+03bbbbb−12.34bbbbb−123

F-25. INPUT. On input, the X specification causes characters to be skipped in the input record. The X field is defined by the presence of an X specification (nX) in the format, where n is the number of characters to be skipped.

Examples:

FORMAT

8X,I2,10X,F4.2,10X,F5.2

INPUT FIELD

WEIGHTbb10bb$1.98bbTOTALbb$19.80

INTERNAL VALUE

10, 1.98, 19.80

F-26. H AND " " SPECIFICATIONS (LITERAL STRINGS)

The H and " " specifications provide for the transfer, without conversion, of a series of ASCII characters (except that quotation marks (") cannot be transferred using " "). A comma after this specification is optional.

F-27. OUTPUT. On output, the ASCII characters in the format specification (there is no associated variable since this is only an editing specification) are output as headings, comments, titles, etc. The specifications are of the form:

$nHc_1c_2 \ldots c_n$ or "$c_1c_2 \ldots c_n$"

where:

n
is the number of characters to be transmitted.

$c_1c_2 \ldots c_n$
are the characters themselves.

H
is the specification type.

" "
is the specification type.

(H alone = 1H; 0H is not permitted.)

Note that with " ", the field length is not specified; it is determined by the number of characters between the quotation marks.

Examples:

FORMAT

20H THIS IS AN EXAMPLE

"THIS ALSO IS AN EXAMPLE"

3"ABC"

RESULT

bTHIS IS AN EXAMPLE

THIS ALSO IS AN EXAMPLE

ABCABCABC

F-28. INPUT. If H is used on input, the number of characters needed to fill the specification is transmitted from the input record to the format. A subsequent output statement will transfer the new heading to the output record. In this way, headings can be altered at run time.

If " " is used on input, the number of characters within the quotation marks is skipped on the input field.

Examples:

FORMAT

31Hbbbbbbbbbbbbbbbbbbbbbbbbbbbbb

INPUT

H INPUT ALLOWS VARIABLE HEADERS

RESULT

31HH INPUT ALLOWS VARIABLE HEADERS

F-29. / SPECIFICATION

The / specification terminates the current record. The / may appear anywhere in the format and need not be set off by commas. Several records may be skipped by preceding the slash with a repetition factor (r-1 records are skipped for r/).

On output, a new record means a new line (list device), a carriage return-linefeed (punch device), or an end-of-record (magnetic tape). Formatted I/O records can be up to 67 words (134 characters) long.

On input, a new record is a new card (card reader), is terminated by a carriage return-linefeed (teleprinter), or is terminated by an end-of-record (magnetic tape).

Note: When the formatter reaches the end of a format and still has values to output, it starts a new record.

Examples:

FORMAT

22x,6HBUDGET///6HWEIGHT,6X,5HPRICE,9X,5HTOTAL,8X

RESULT

(line 1) bbbbbbbbbbbbbbbbbbbbbbBUDGET

(line 2)

(line 3)

(line 4) WEIGHTbbbbbbPRICEbbbbbbbbbTOTALbbbbbbbb

F-30. HOW TO PUT FORMATS TOGETHER

Formats can be specified together as follows:

1. When two specifications follow each other they are concatenated.

   Format: E9.4,I6

E field	I field
9 characters	6 characters

2. To leave space between numbers use X.

   Format: E9.4,3X,I6

E field	X	I field
9 characters	3 characters	6 characters

3. To start a new line, use /

   Format: E9.4/I6

E field
9 characters

I field
6 characters

4. Specifications can be gathered together into groups and surrounded by parentheses.

   Example: (E9.3,2X,I6)

E	X	I

These groups can be nested one level deep.

For example,

(E9.3,3(2X,I6))

E	X	I	X	I	X	I

(E9.3,3(2X,I6)2X2(I8))

E	X	I	X	I	X	I	X	I	X	I	

5. Use the repetition factor to repeat single specifications (except nH) or groups of specifications. This is done by preceding the specification or parenthetical groups with a repeat count, r. The conversion is repeated up to r times, unless the list of variables is exhausted first.

   3(E9.3,2X,I6,2X)/

E	X	I	X	E	X	I	X	E	X	I	X

E	X	I	X	E	X	I	X	E	X	I	X

6. Use the principle of unlimited groups — when the formatter has exhausted the specifications of a format and still has list items left, it inputs a new record for a READ or outputs the present record for a WRITE and returns to the last, outer-most unlimited group within the format. An unlimited group is a set of specifications enclosed in parentheses. If the format has no unlimited groups, the formatter returns to the beginning of the format.

   Example: Format = (I5,2(3X,F8.4,8(I2))

   Format = (I5,2(3X,F8.4,8(I2I2)),4X,3(I6))

   Format = (I5,3X,4F8.4,3X)

7. Keep in mind the accuracy limitations of your data. Although the formatter will print out or read in as many digits as specified, only certain digits are significant:

   Integer variables can be between $-32{,}768_{10}$ and $+32{,}767_{10}$.

   Floating-point numbers can guarantee 6 digits of accuracy (plus exponent).

8. On input, blanks are not evaluated as part of the data item.

F-31. FREE-FIELD INPUT

When free-field input is used, a format specification is not used. Special symbols are included within the input data to direct the conversion process.

For example,

Space or comma	data item delimiters.
/	record terminator.
+ −	sign of item.
. E + − D	floating-point number.
@	octal integer.
". . ."	comments.

All other ASCII non-numeric characters are treated as spaces (and delimiters). Free-field input may be used for numeric data only.

F-32. DATA ITEM DELIMITERS

Any contiguous string of numeric and special formatting characters occurring between two commas, a comma and a space, or two spaces, is a data item whose value corresponds to a list element. A string of consecutive spaces is equivalent to one space. Two consecutive commas indicate that no data item is supplied for the corresponding list element; the current value of the list element is unchanged. An initial comma causes the first list element to be skipped.

Examples:

1. Input data : 1720, 1966, 1980, 1392

   Result in Memory: 1720
   1966
   1980
   1392

2. Input Data: 1266, , 1794, 2000

   Result in Memory: 1266
   1966
   1794
   2088

F-33. FLOATING-POINT INPUT

The symbols used to indicate a floating-point data item are the same as those used in representing floating-point data for format specification directed input, as follows:

Integer field — Fraction field — Exponent field

±n. . .n.n. . .n±Eee

Decimal point

If the decimal point is not present, it is assumed to follow the last digit. For example, 3.14, 314E−2, 3140E−3, .0314+2 and .314E1 are all equivalent to 3.14.

F-34. OCTAL INPUT

An octal input item has the following format:

$@x_1 . . . x_d$

The symbol @ defines an octal integer. The x's are octal digits each in the range of 0 through 7. List elements corresponding to the octal data items must be type integer.

F-35. RECORD TERMINATOR

A slash within a record causes the next record to be read as a continuation of the data list. The remainder of the current record is skipped as comments.

Example:

INPUT DATA

987, 654, 321, 123/DESCENDING

456

RESULT IN MEMORY

987 654 321 123 456

F-36. COMMENTS WITHIN INPUT

All characters appearing between a pair of quotation marks in the same line are considered to be comments and are ignored.

For example,

"6.7321" is a comment and is ignored.

6.7321 is a real number and is stored as such.

F-37. EXAMPLE CALLING SEQUENCES

Example 1: Unformatted output Example

PURPOSE

1000 2-word elements in the array ARRAY are punched on the standard punch unit. The output will consist of 60 word records (59 data words and 1 control word) until the entire array is punched.

```
        LDA  PUNCH     Output unit number
        CLB            Output flag
        JSB  .BIO.     Binary initialization entry
                       point
        LDA  =D1000    Number of elements in array
        LDB  ADRES     Location of array
        JSB  .RAR.     Real (2-word) array entrance
        JSB  .DTA.     Output termination
        →
          .
          .
          .
PUNCH   DEC  4         Unit number
ADRES   DEF  ARRAY     Location of ARRAY
ARRAY   BSS  2000      Defines 1000 2-word elements.
```

Example 2: Internal Conversion and Free-Field Input

PURPOSE

The ASCII data starting at BUFFR is converted in free field form to binary. R will contain the binary representation of .0001234 and I will contain the binary representation of 28.

```
        CLA                Internal conversion
                           flag
        CLB,INB            ASCII to binary flag
        JSB     .DIO.      Initialization
        DEF     BUFFR      Location of ASCII data
        ABS     0          Specifies ASCII data is
                           in free-field form
        DEF     ENDLS      End of list
        JSB     .IOR       Declare real variable
        DST     R          Store binary item in R
        JSB     .IOI.      Declare integer vari-
                           able
        STA     I          Store in I
ENDLS   →
          .
          .
          .
R       BSS     2          Real variable
I       BSS     1          Integer variable
BUFFR   ASC     6,123.4E-6,28  ASCII data to be con-
                           verted to binary.
```

F-38. INTERNAL CONVERSION

The formatter provides the programmer with the option of using the conversion parts of the formatter only, without any input or output. This process is called "internal conversion."

On input, ASCII data is read from a buffer and converted according to a format (or free-field) into a variable list (this is known as decoding). On output, binary data is converted to ASCII according to a format and stored in a buffer (this is known as encoding).

Internal conversion ignores "/" specifications or unlimited groups. The concept of records does not apply during internal conversion.

OUTPUT CALLING SEQUENCE (BINARY TO ASCII CONVERSION): ENCODING

```
CLA
CLB
JSB  .DIO.
DEF  buffer (destination)
DEF  format
DEF  end of list
  .
  .
Calls  to define each variable (same as regular calls)
  .
  .
Termination Call
      (Same as regular calls)
```

where *buffer* is a storage area for the ASCII output.

INPUT CALLING SEQUENCE (ASCII TO BINARY CONVERSION): DECODING

Formatter	*Free Field*
CLA	CLA
CLB, INB	CLB, INB
JSB .DIO.	JSB .DIO.
DEF *buffer*	DEF *buffer*
DEF *format*	ABS 0
DEF *end of list*	*DEF end of list*
.	.
.	.

Calls to define each variable (Same as regular calls)

where buffer is a storage area containing ASCII characters which will be converted by the formatter into binary values.

F-39. BUFFERED I/O WITH THE FORMATTER

Normally, when a program uses the formatter, it can execute only one I/O operation at a time. The internal conversion feature of the formatter, however, can be used with direct calls to .IOC. to provide both buffered and formatter I/O.

The flowchart in figure F-3 shows how a program can read in data from two units (U1 and U2) into two buffers (B1 and B2) at the same time by calling .IOC. When unit U1 is complete, buffer B1 is converted into list L1 by the formatter (while input continues on unit U2).

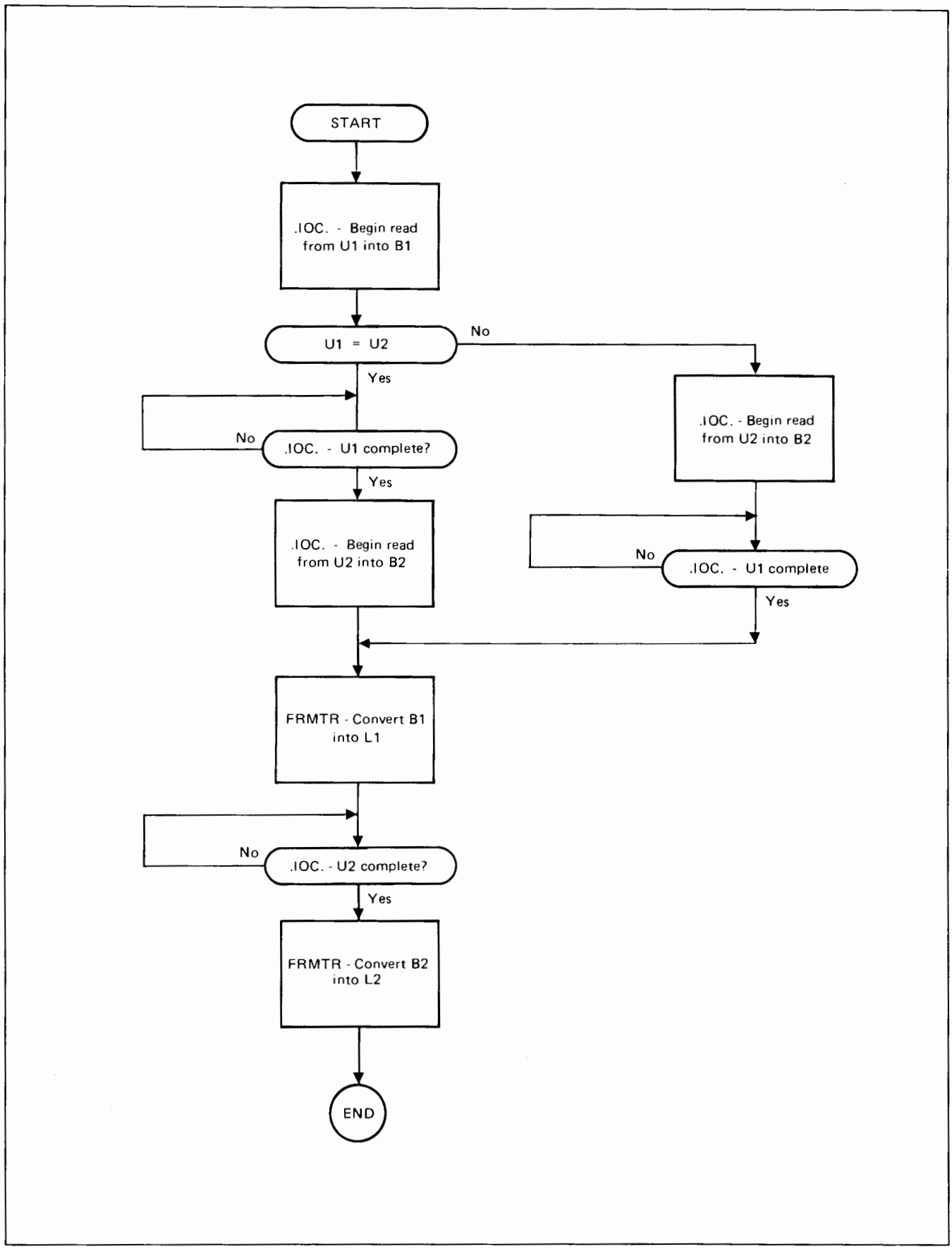


Figure F-3. Buffered I/O With the Formatter

ASSEMBLER ERROR MESSAGES

APPENDIX G

Errors detected in the source program are indicated by a 1- or 2-letter mnemonic followed by the sequence number and the first 62 characters of the statement in error. The messages are printed on the list output device during the passes indicated. A message specifying the number of errors detected is printed on the system console device at the end of each pass.

Error listings produced during Pass 1 are preceded by a number which identifies the source input file where the error was found. Pass 2 error messages are preceded by a reference to the previous page of the listing where an error message was written. The first error will refer to page "0". The error count at the end of Pass 2 is preceded by the page number in the listing where the final error was encountered.

Error Code	Pass	Description
CS	1	Control statement error: a) The control statement contained a parameter other than the legal set. b) Both A and R were specified. c) There was no output parameter (B, T, or L) and the Job Binary parameter was not specified.
DD	1	Doubly defined symbol: A name defined in the symbol table appears more than once as: a) A label of a machine instruction. b) A label of one of the pseudo operations: BSS DBL BYT DBR ASC EQU DEC ABS DEF OCT DEX Arithmetic subroutine call c) A name in the Operand field of a COM or EXT statement. d) A label in an instruction following a REP pseudo operation. e) Any combination of the above. An arithmetic subroutine call symbol appears in a program both as a pseudo instruction and as a label.
EN	1	The symbol specified in an ENT statement has already been defined in an EXT statement.
EN UNDEF <symbol>	2	The entry point specified in an ENT statement does not appear in the label field of a machine or BSS instruction. The entry point has been defined in the Operand field of an EXT statement (or has been equated to an absolute value of zero — this is not an error, but is noted).

Error Code	Pass	Description
IF	1	An IFZ or an IFN follows either an IFZ or an IFN without an intervening XIF. The second pseudo instruction is ignored.
IL	1	Illegal instruction: a) Instruction mnemonic cannot be used with type of assembly requested in control statement. The following are illegal in an absolute assembly: NAM EXT ENT COM Arithmetic subroutine calls b) The ASMB statement has an R parameter, and NAM has been detected after the first valid Opcode.
	1 or 2	Illegal character: A numeric term used in the Operand field contains an illegal character (e.g. an octal constant contains other than +, −, or 0-7). This code may also appear following an M error for missing operands.
LB	1	Missing label in an EQU or RPL pseudo instruction.
M	1 or 2	Illegal operand: a) Operand is missing for an Opcode requiring one. b) Operands are optional and omitted but comments are included for: END HLT c) Operand is an external symbol or an indirect address for: DBL DBR d) An absolute expression in one of the following instructions from a relocatable program is greater than 1777_8. Instructions referencing memory locations DEF, DBL, and DBR Arithmetic subroutine calls e) A negative operand is used with an Opcode other than ABS, DEX, DEC, OCT, and BYT. f) A character other than I follows a comma with operands which can be indirect. g) Operand is an indirect address when used with JPY. h) Using a literal as the second operand in the following instructions: TBS SBS CBS i) A character other than C follows a comma in certain I/O instructions. j) A relocatable expression in the Operand field of one of the following: ABS ASR RRL REP ASL LSR SPC RRR LSL k) An illegal operator appears in an Operand field (e.g. + or − as the last character). l) An ORG statement appearing in a relocatable program includes an expression that is common relocatable or absolute. m) A relocatable expression contains a mixture of program and common relocatable terms. n) An external symbol appears in an operand expression or is specified as indirect.

Error Code	Pass	Description
		o) The literal, literal code, or type of literal is illegal for the operation code used (e.g., STA = B7). p) An integer expression in one of the following instructions does not meet the condition $1 \leq n \leq 16$. The integer is evaluated modulo 2^4. ASR RRR LSR ASL RRL LSL q) The value of an 'L' type literal is relocatable. r) The number of words, n, specified for an ASCII string definition, ASC n, exceeds 28 decimal words.
NO	1 or 2	No origin definition: The first statement in the assembly containing a valid opcode following the ASMB control statement (and remarks and/or HED, if present) is neither an ORG nor a NAM statement. If absolute, the program is assembled starting at 2000; if relocatable, the program is assembled starting at zero.
OP	1 or 2	Illegal Opcode preceding first valid Opcode. The statement being processed does not contain an asterisk in position one. The statement is assumed to contain an illegal Opcode; it is treated as a remarks statement. Illegal Opcode: A mnemonic appears in the Opcode field which is not valid. A word is generated in the object program.
OV	1 or 2	Numeric operand overflow: The numeric value of a term or expression has overflowed its limit: $1 \geq N \geq 16$ Shift-Rotate Set 2^2-1 Input/Output, Overflow, Halt $2^{10}-1$ Memory Reference (in absolute assembly) 2^{15} Data generated by DEC or DEX $2^{15}-1$ DEF and ABS operands and expressions concerned with program location counter. $2^{16}-1$ OCT
SO		There are more symbols defined in the program than the symbol table can handle.
SY	1 or 2	Illegal Symbol: A Label field contains an illegal character or is greater than 5 characters. A label with illegal characters may result in an erroneous assembly if not corrected. A long label is truncated on the right to 5 characters. Illegal Symbol: A symbolic term in the Operand field is greater than five characters; the symbol is truncated on the right to 5 characters. Too many control statements: The source file contains more than one control statement. The Assembler assumes that the second control statement is a label, since it begins in column 1. Thus, the commas are considered as illegal characters and the "label" is too long. The binary object program is not affected by this error. The first control statement processed is the one used by the Assembler.
UN	1 or 2	Undefined Symbol: a) A symbolic term in an Operand field is not defined in the Label field of an instruction or is not defined in the Operand field of a COM or EXT statement. b) A symbol appearing in the Operand field of one of the following pseudo operations was not defined previously in the source program: BSS ASC EQU ORG END

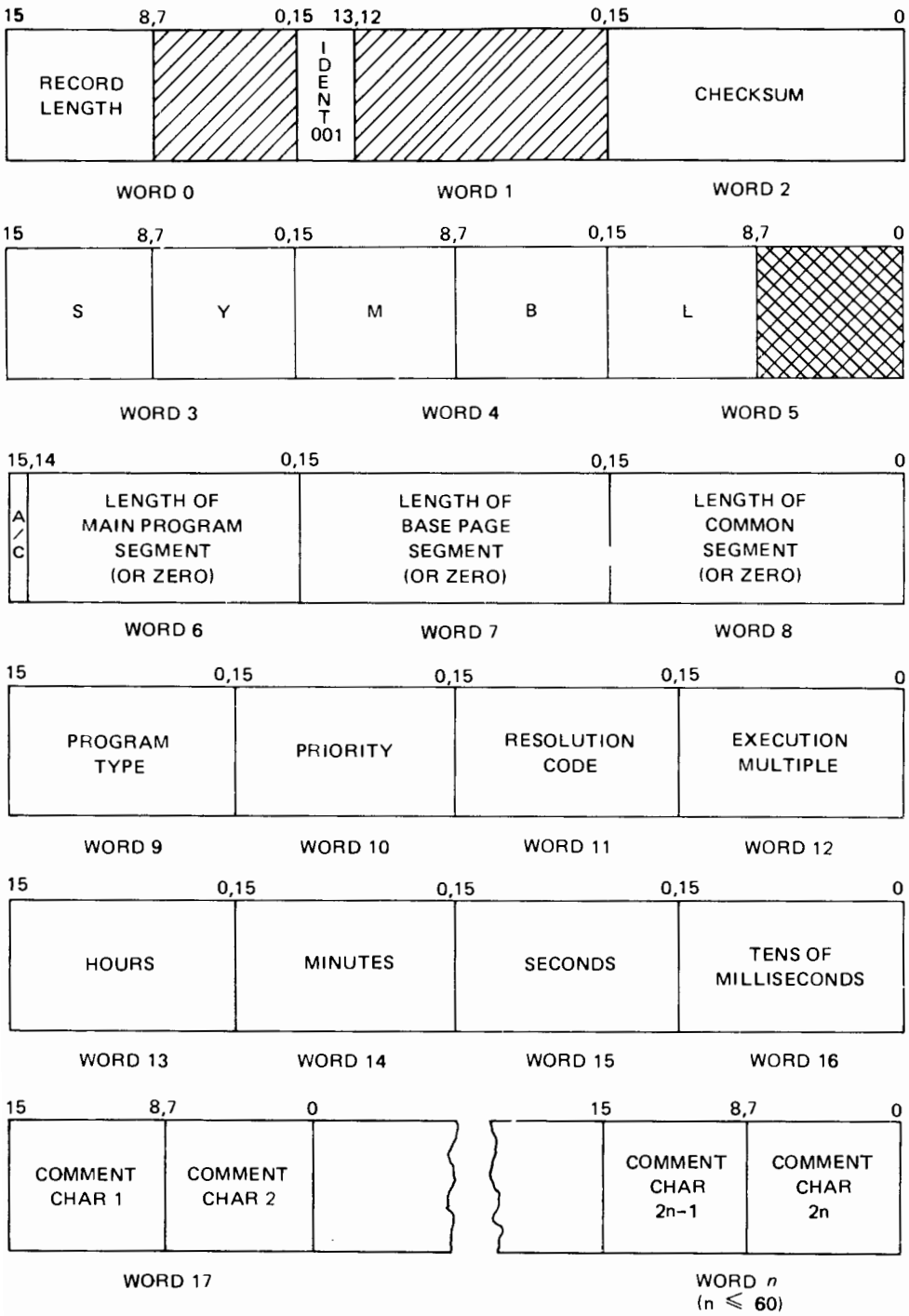
TAPE FORMATS

APPENDIX H

NAM RECORD

CONTENT

EXPLANATION



RECORD LENGTH = 9-60 WORDS

IDENT = 001

CHECKSUM: ARITHMETIC TOTAL OF ALL WORDS IN RECORD EXCLUDING WORDS 1 AND 3.

SYMBL: FIVE CHARACTER NAME OF PROGRAM

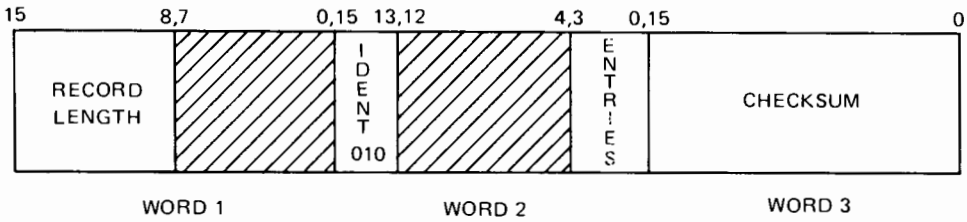
A/C: BINARY TAPE PRECESSION
= 0 IF ASSEMBLER PRODUCED OR LENGTH IS EXACT
= 1 IF COMPILER PRODUCED OR LENGTH IS UNKNOWN

HATCH-MARKED AREAS SHOULD BE ZERO-FILLED WHEN THE RECORDS ARE GENERATED

CROSS-HATCH-MARKED AREAS SHOULD BE SPACE FILLED WHEN THE RECORDS ARE GENERATED

ENT RECORD

CONTENT

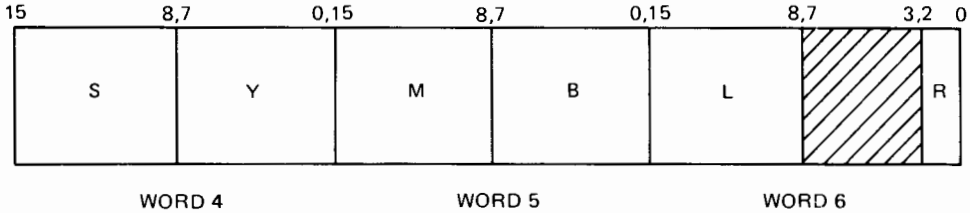


EXPLANATION

RECORD LENGTH = 7-59 WORDS

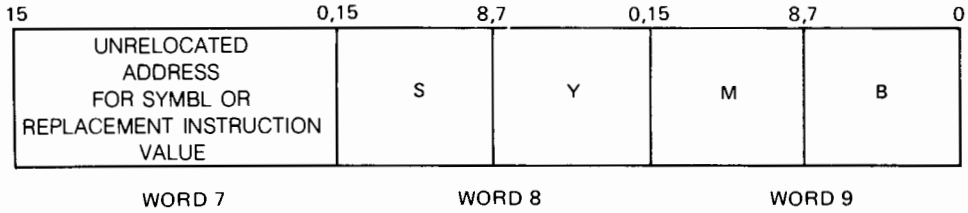
IDENT = 010

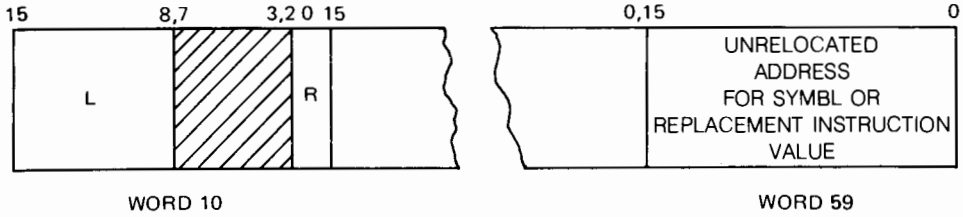
ENTRIES: 1 TO 14 ENTRIES PER RECORD; EACH ENTRY IS FOUR WORDS LONG.



SYMBL: 5 CHARACTER ENTRY POINT SYMBOL

R: RELOCATION INDICATOR
= 0 IN PROGRAM RELOCATABLE
= 1 IF BASE PAGE RELOCATABLE
= 2 IF COMMON RELOCATABLE
= 3 IF ABSOLUTE
= 4 INSTRUCTION REPLACEMENT

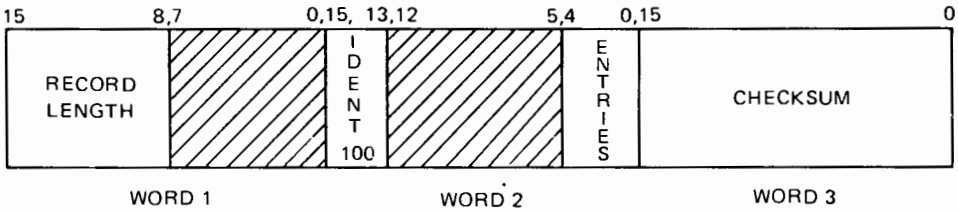


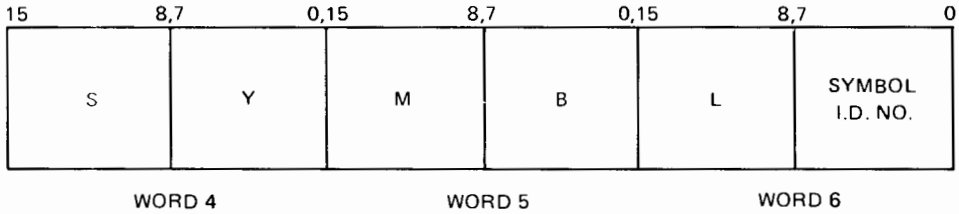


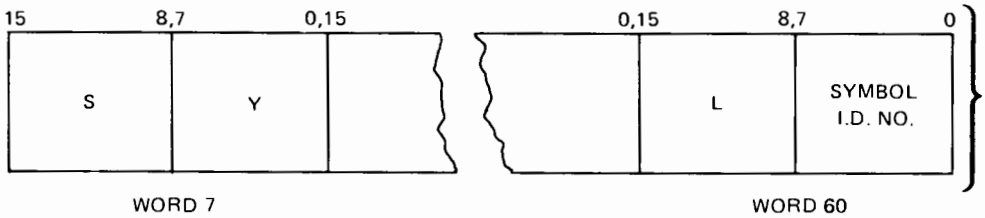
WORDS 4 THROUGH 7 ARE REPEATED FOR EACH ENTRY POINT SYMBOL.

EXT RECORD

CONTENT







EXPLANATION

RECORD LENGTH = 6-60 WORDS

IDENT = 100

ENTRIES: 1 TO 19 PER RECORD; EACH ENTRY IS THREE WORDS LONG

SYMBL: 5 CHARACTER EXTERNAL SYMBOL

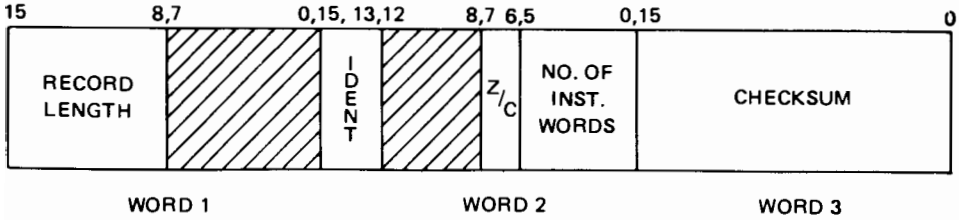
SYMBOL ID. NO.: NUMBER ASSIGNED TO SYMBL FOR USE IN LOCATING REFERENCE IN BODY OF PROGRAM.

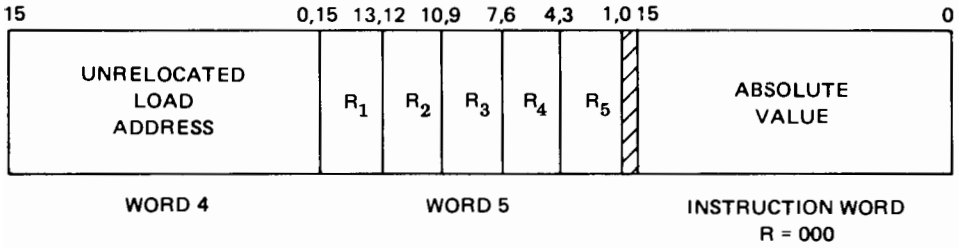
WORDS 4 THROUGH 6 REPEATED FOR EACH EXTERNAL SYMBOL (MAXIMUM OF 19 PER RECORD).

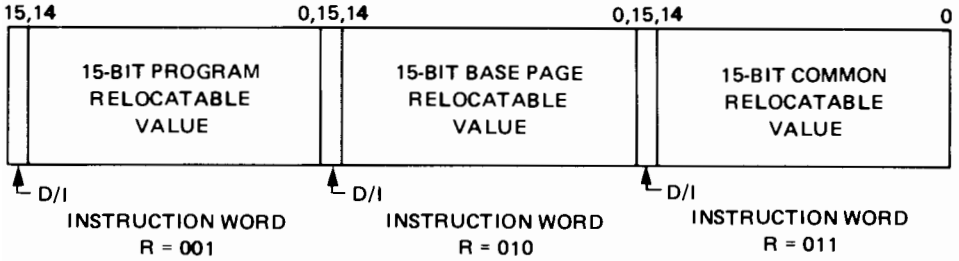
Compute! Museum

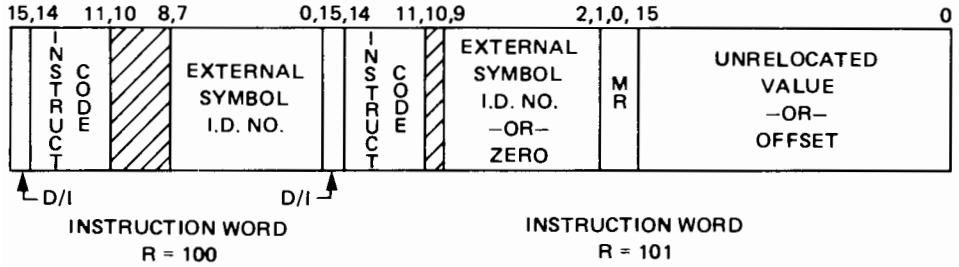
DBL RECORD

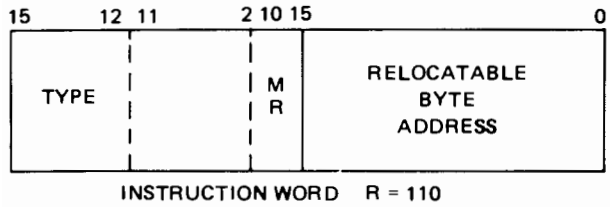
CONTENT











EXPLANATION

RECORD LENGTH = 6-60 WORDS
IDENT = 011
Z/C: RELOCATION OF LOAD ADDRESS
= 0 FOR BASE PAGE
= 1 FOR PROGRAM
= 2 FOR ABSOLUTE
= 3 FOR COMMON
NO. OF INST. WORDS: 1 TO 45 LOADABLE INSTRUCTION WORDS PER RECORD

RELOCATABLE LOAD ADDRESS: STARTING ADDRESS FOR LOADING THE INSTRUCTIONS WHICH FOLLOW;

R's: RELOCATION INDICATORS:
000 = ABSOLUTE
001 = 15-BIT PROGRAM RELOCATABLE
010 = 15-BIT BASE PAGE RELOCATABLE
011 = 15-BIT COMMON RELOCATABLE
100 = EXTERNAL REFERENCE
101 = MEMORY REFERENCE
110 = BYTE ADDRESS

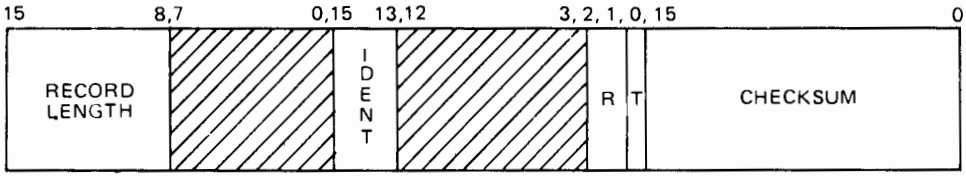
R_1 IS RELOCATION INDICATOR FOR INSTRUCTION $WORD_1$; R_2, FOR INSTRUCTION $WORD_2$; ETC.

D/I: INDIRECT ADDRESSING
0 = DIRECT
1 = INDIRECT

MEMORY REFERENCE INSTRUCTIONS USE TWO WORDS, WITHIN THE TWO-WORD GROUP "MR" INDICATES RELOCATABILITY OF OPERAND SPECIFIED IN SECOND WORDS:
00 = PROGRAM RELOCATABLE
01 = BASE PAGE RELOCATABLE
10 = COMMON RELOCATABLE
11 = ABSOLUTE

END RECORD

CONTENT



WORD 1 WORD 2 WORD 3

WORD 4

EXPLANATION

RECORD LENGTH = 4 WORDS
IDENT = 101

R: RELOCATION INDICATOR FOR TRANSFER ADDRESS

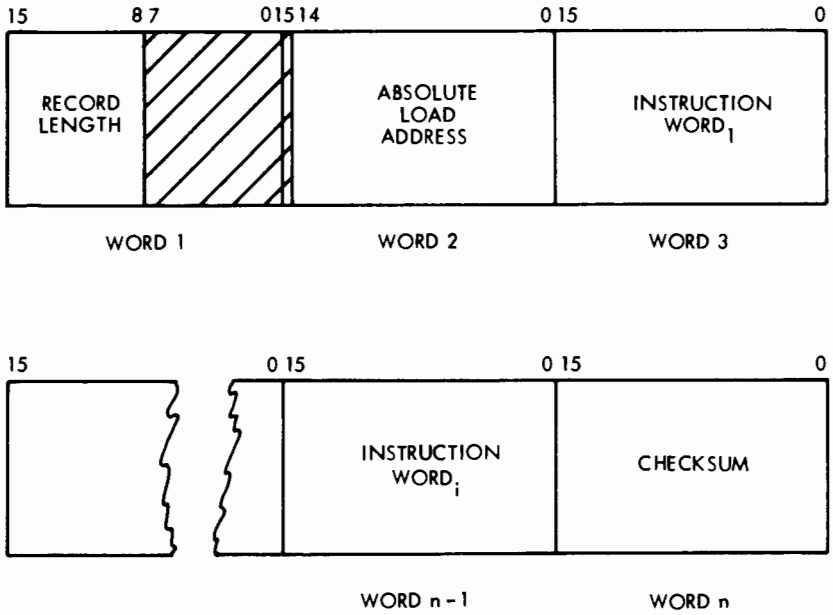
= 0 IF PROGRAM RELOCATABLE
= 1 IF BASE PAGE RELOCATABLE
= 2 IF COMMON RELOCATABLE
= 3 IF ABSOLUTE

T: TRANSFER ADDRESS INDICATOR

= 0 IF NO TRANSFER ADDRESS IN RECORD
= 1 IF TRANSFER ADDRESS PRESENT

ABSOLUTE TAPE FORMAT

CONTENT



EXPLANATION

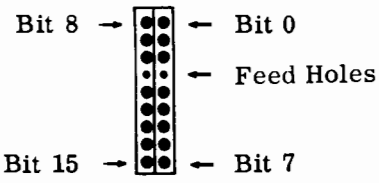
RECORD LENGTH = NUMBER OF WORDS IN RECORD EXCLUDING WORDS 1 AND 2 AND THE LAST WORD.

ABSOLUTE LOAD ADDRESS: STARTING ADDRESS FOR LOADING THE INSTRUCTIONS WHICH FOLLOW

INSTRUCTION WORDS: ABSOLUTE INSTRUCTIONS OR DATA

CHECKSUM: ARITHMETIC TOTAL OF ALL WORDS EXCEPT FIRST AND LAST

On paper tape, each word represents two frames arranged as follows:



RTE CROSS REFERENCE TABLE GENERATOR

APPENDIX I

This Real-Time Executive Operating System Cross Reference Table Generator routine (XREF) processes an assembler source program and provides a list of all symbols and symbol references used within the program.

I-1. COMPUTER CONFIGURATION

The routine requires a Real-Time Executive Operating System with logical unit 1 as the system console and a standard list device.

I-2. FUNCTIONAL AND OPERATIONAL CHARACTERISTICS

Source program input may come from:

a. An input unit specified by a logical unit number,

b. 'EDITR'-CREATED FILE (Logical Source Pointer must be set) or,

c. The temporary work area of the disc which was set up by the Assembler in a previous assembly.

I-3. OUTPUT FORMAT

The general format of the output list is:

```
SSSSS DDDDD/TT RRRRR /TT  . . . RRRRR /TT
      RRRRR ;TT           . . . RRRRR  /TT
```

where:

SSSSS
is the symbol which may be any legal label to the assembler.

DDDDD
is the statement number in decimal in which the label was defined. It has a maximum value of 32767, when using the no tape number option, and a maximum value of 2047 when using tape numbers.

TT
is the decimal tape or file number (following a zero length record) with a maximum value of 16.

RRRRR
is the statement number in decimal in which the label was referenced. It has the same physical limits as the defining statement numbers.

Note: The defined format and meaning of SSSSS, DDDDD, and TT are used in the following paragraphs.

I-4. PSEUDO PROCESSING

ORG, ORB, ORR, IFN, IFZ, XIF, MIC, and MIC-defined OPCODES are listed as:

```
**ORG   ********  RRRRR/TT,   . . .  RRRRR/TT
**ORB     "          "                  "
**ORR     "          "                  "
**IFN     "          "                  "
**XIF     "          "                  "
**MIC     "          "                  "
**XYZ     "          "                  "
```

(where 'XYZ' is a 'MIC'-defined opcode.)

The defining statement number is replaced by a string of asterisks.

I-5. DOUBLE DEFINED PROCESSING

If a symbol has been defined more than once, it is listed in the symbol list in the following format:

```
SSSSS   ########   RRRRR/TT . . . . RRRRR/TT
```

where:

SSSSS
is the symbol.

I-6. UNDEFINED LABEL PROCESSING

A symbol is referenced but not defined. The entry in the symbol list has the following format:

```
SSSSS   ???????   RRRRR/TT,   . . .   RRRRR/TT
```

The defining statement number is replaced by question marks.

I-7. UNUSED LABEL PROCESSING

If a symbol is defined but not referenced by a statement, the entry in the symbol list has the following format:

@SSSSS DDDDD/TT

The symbol is preceded by a "@".

I-8. LITERAL PROCESSING

If a literal of the type =L is referenced by a statement, the characters following the =L are handled as a normal symbol.

If a literal of the type =A, =B, =D, or =F is referenced by a statement, the symbol list has the following format:

LLLLL ..... RRRRR/TT, ... RRRRR/TT

where:

LLLLL
is an exact copy of the literal. DDDDD, the defining statement number, is replaced by dots.

If the literal is seven or more characters long: LLLLL is a maximum length of seven characters, the defining statement number does not have the first dot, and only the first seven characters are used. For example, =B12345 and =B123456 would be considered as the same literal =B12345 and would have the format.

=B12345 ..... RRRRR/TT, ... RRRRR/TT

I-9. OPERATION DIRECTIVE

*ON,XREF,A [,B,C,D,E]

where:

A
is input area or device.

A = 2 input from SOURCE file of disc defined by LS directive *LS,LU,START TRACK

= N input from logical unit device N

= −LU/ST Assembler supplied code for RTE Assembler-scheduled processing ('C' control statement option).

B
is character limits specification

B = 0 to specify no limits

= N to request the entering of limits from the teleprinter and to allow multiple passes of the cross reference routine.

C
is tape number or no tape number specification corresponding to the mode used in the Assembly.

C = 0 to specify use of tape numbers and a tape length less than 2048 statements

Note: A blank card inserted into a card deck before statement 2048 indicates an end-of-tape.

= N to specify no tape numbers are to be used (sequence numbers can be as large as 32767).

= −N XREF will number pages consecutively from the last RTE-ASMB page number (−N = −page no.)

D
is the number of lines per page

D = 0 to print 57 lines per page

= N to print N lines per page

E
is the list device LU #

E = 0 default to LU #6

E = N output to LU #N

Note: The cross reference routine can also be requested to run immediately after an assembly. XREF can be specified via a "C" parameter in the ASMB control statement. When the XREF routine is scheduled via the "C" parameter, the following options are assumed: B = 0, C = 0, D = 57, E = Assembler list parameter.

I-10. BOUNDS

If the second parameter of the ON, XREF directive is non-zero, this message is printed,

/XREF: ENTER LIMITS <LH> OR </E> ?

enter the XREF bound limits from the system teleprinter using this format:

LH (CR) (LF) or /E (CR) (LF)

where:

L
is lower bound character. (Lowest is a blank, space bar on teleprinter)

H
is upper bound character. (Highest is a left arrow←)

/E
indicates XREF limit termination and exit to system.

Table I-1 contains a list of messages (and their meanings) the user may receive using the ON,XREF directive. Enter the following format to receive these messages:

/XREF: <*message*>

Table I-1. XREF Messages

MESSAGE	MEANING
END OF FILE	End of a user specified source file is reached before an END instruction is found. The XREF routine terminates.
TABLE OVERFLOW	XREF routine does not have enough core space for the table entries. The XREF routine can be made to run with the option for specifying lower and upper bounds and use of multiple passes.
$END <******>	Termination Message. Absolute assembly sources appear as shown. Relocatable sources contain NAM symbols: <NAME>.
ENTER LIMITS OR /E	A request is made to enter XREF bound limits from the system teleprinter.
SOURCE LS?	The Logical Source is not defined, or for RTE: A = 0. The XREF routine terminates.
>16 TAPES!!	More than 16 tapes or zero-length records have been encountered. XREF terminates.

I-11. SAMPLE CROSS-REFERENCE GENERATION

The following pages show a sample Assembler program using cross-reference generation.

```
 PAGE 0001


0001                    ASMB,R,B,L,T,Z,C

 # 002
DD  0019  THREE OCT 3
START R 000000
AGAIN R 000002
LOOP  R 000005
NEXT  R 000011
NOUSE R 000012
ADD   R 000013
ADDR  R 000016
TIMES R 000017
THREE B 000000
TWO   B 000001
INIT  R 000020
COUNT R 000026
ONE   R 000027
DNUM  B 000002
NUM   R 000031
HERE  R 000043
**0001 ERRORS PASS#1 **RTE ASMB 750420**
```

```
 PAGE 0002 #01


0001                    ASMB,R,B,L,T,Z,C
0002  00000                     NAM EXAMP      DO-NOTHING USEFUL
0003  00000 000000  START NOP
0004  00001 016020R       JSB INIT
0005  00002 062017R AGAIN LDA TIMES
0006  00003 072026R       STA COUNT      SET COUNTER FOR LOOP
0007  00004 062043R       LDA *D123      INITIALIZE FIRST VALUE
0008  00005 172016R LOOP  STA ADDR,I     SAVE VALUE
0009  00006 042044R       ADA *B123456 CALCULATE NEXT VALUE
0010  00007 036016R       ISZ ADDR
0011  00010 036026R       ISZ COUNT      BUMP COUNT
0012  00011 026005R NEXT  JMP LOOP       REPEAT UNTIL DONE
0013  00012 062017R NOUSE LDA TIMES
0014                      IFN
0015                ADD   ADA ONE        ONE IF BY 'N'
0016                      XIF
0017                      IFZ
0018  00013 040001B ADD   ADA TWO        TWO IF BY 'Z'
0019                      XIF
0020  00014 072017R       STA TIMES
0001  00015 026002R       JMP AGAIN      SECOND TAPE
0002  00011               ORG NEXT
0003  00011 026002R       JMP AGAIN
0004  00016               ORR
0005  00016 000000  ADDR  NOP
0006  00017 000000  TIMES NOP
0007  00000               ORB
0008  00000 000003  THREE DEC 3
0009  00001 000002  TWO   DEC 2
0010  00020               ORR
0011  00020 000000  INIT  NOP
0012  00021 060002B       LDA DNUM
0013  00022 072016R       STA ADDR

PG 000
UN  0014        LDA NEG10
0014  00023 062000        LDA NEG10
0015  00024 072017R       STA TIMES
0016  00025 126020R       JMP INIT,I
0017  00026 000000  COUNT NOP
0018  00027 000001  ONE   OCT 1
0019  00030 000003  THREE OCT 3
0020  00002               ORB
0021  00002 000031R DNUM  DEF NUM
0022  00031               ORR
0023  00031 000000  NUM   BSS 10
0024  00043         HERE  EQU *
      00043 000173
      00044 123456
0025                      END START

PG 002
**0002 ERRORS *TOTAL **RTE ASMB 750420**
```

```
PAGE 0003                    EXAMP       DO-NOTHING USEFUL

                    CROSS-REFERENCE SYMBOL TABLE

**IFN   ******** 00014/01

**IFZ   ******** 00017/01

**ORB   ******** 00007/02 00020/02

**ORC   ******** 00002/02

**ORR   ******** 00004/02 00010/02 00022/02

**XIF   ******** 00016/01 00019/01

=B12345 ........ 00009/01

=D123   ........ 00007/01

ADD     ######## 00015/01 00018/01

ADDR    00005/02 00008/01 00010/01 00013/02

AGAIN   00005/01 00001/02 00003/02

COUNT   00017/02 00006/01 00011/01

DNUM    00021/02 00012/02

*HERE   00024/02

INIT    00011/02 00004/01 00016/02

LOOP    00008/01 00012/01

NEG10   ???????? 00014/02

NEXT    00012/01 00002/02

*NOUSE  00013/01

NUM     00023/02 00021/02

ONE     00018/02 00015/01

START   00003/01 00025/02

THREE   ######## 00008/02 00019/02

TIMES   00006/02 00005/01 00013/01 00020/01 00015/02

TWO     00009/02 00018/01
```

Computer Museum

INDEX

ABS, 4-12, B-9, C-1
Absolute Expressions, 2-4
ADA, 3-1, B-2, C-1
ADB, 3-1, B-2, C-1
Add Instructions, 3-1
Address Definition Pseudo Instruction, 4-11
Address Expressions, 2-4
Addressing
  Indirect, 2-6
  Memory, 1-1
  Symbolic, 1-1
ADX, 3-7, B-5, C-1
ADY, 3-7, B-5, C-1
ALF, 3-4, B-3, C-1
Alphabetic List of Instructions, C-1
ALR, 3-4, B-3, C-1
ALS, 3-4, B-3, C-1
Alter-Skip Instructions, 3-4
AND, 3-2, B-2, C-1
Arithmetic Subroutine Calls, 4-20
ARS, 3-4, B-3, C-1
ASC, 4-14, B-10, C-1
ASL, 3-10, B-6, C-1
ASMB Statement, 1-3
ASR, 3-9, B-6, C-1
Assembler Control Pseudo Instructions, 4-1
Assembler Error Messages, G-1
Assembly Listing Control Pseudo Instructions, 4-19
Assembly Options, 1-3
Asterisk, 2-2, 2-3, 2-4

BCD-ASCII Conversion, A-4
Binary Output, 1-3
Bit Processing Instructions, 3-3
BLF, 3-4, B-3, C-1
BLR, 3-4, B-3, C-1
BLS, 3-4, B-3, C-1
Bounds, I-2
BRS, 3-4, B-3, C-1
BSS, 4-19, B-10, C-1
BYT, 4-19, B-10, C-1
Byte Processing Instructions, 3-2

CAX, 3-5, B-5, C-1
CAY, 3-5, B-5, C-1
CBS, 3-4, B-3, C-1
CBT, 3-3, B-3, C-1
CBX, 3-5, B-5, C-1
CBY, 3-5, B-5, C-1
CCA, 3-4, B-4, C-1
CCB, 3-4, B-4, C-1
CCE, 3-4, B-4, C-1
Character Set, HP Computer Systems, 2-1, A-1
CLA, 3-4, B-4, C-1
CLB, 3-4, B-4, C-1
CLC, 3-8, B-6, C-1
CLE, 3-4, B-3, B-4, C-1
Clear Flag Indicator, 2-7
CLF, 3-8, B-6, C-1
CLO, 3-9, B-6, C-1
CMA, 3-4, B-4, C-1
CMB, 3-4, B-4, C-1
CME, 3-4, B-4, C-1
CMW, 3-2, B-2, C-1
COM, 4-5, B-9, C-1
Comments Field, 2-7
Computer Configuration, I-1
Consolidated Coding Sheets, D-1
Constant Definition Pseudo Instructions, 4-14
Control Statement, 1-3
Conversion, Internal, F-10
Counter, Program Location, 1-3
CPA, 3-2, B-2, C-1
CPB, 3-2, B-2, C-1
Cross-Reference Table (XREF), RTE, I-1
CXA, 3-5, B-5, C-1
CXB, 3-5, B-5, C-1
CYA, 3-5, B-5, C-1
CYB, 3-5, B-5, C-1

DBL, 4-13, B-10, C-1
DBR, 4-13, B-10, C-1
DEC, 4-14, B-10, C-1
DEF, 4-11, B-9, C-1
Define User Instruction Pseudo Instruction, 4-21
Delimiters, Field, 2-1
DEX, 4-17, B-10, C-1
DIV, 3-9, B-6, C-1
DJP, 3-12, B-7, C-1
DJS, 3-12, B-7, C-1
DLD, 3-9, B-6, C-1
DST, 3-9, B-6, C-1
DSX, 3-6, B-5, C-1
DSY, 3-6, B-5, C-1
Dynamic Mapping System, 3-10

EAU Instructions, 3-9
ELA, 3-4, B-3, C-1
ELB, 3-4, B-3, C-1
END, 4-5, B-9, C-1
ENT, 4-7, B-9, C-1
ERA, 3-4, B-3, C-1
ERB, 3-4, B-3, C-1
Error Messages, Assembler, G-1
EQU, 4-13, B-10, C-1
Evaluation of Expressions, 2-4
Expression Operators, 2-4
Expression Terms, 2-4
Expressions
  Absolute, 2-4
  Evaluation of, 2-4
  Relocatable, 2-4

EXT, 4-7, B-9, C-1
Extended Arithmetic Unit Instructions, 3-9

FAD, 3-10, B-7, C-1
FDV, 3-10, B-7, C-1
Fences, 3-18
Field Delimiters, 2-1
FIX, 3-10, B-7, C-1
Flag, I/O Interrupt, 2-7
Floating Point Instructions, 3-10
FLT, 3-10, B-7, C-1
FMP, 3-10, B-7, C-1
Formatter, The, F-1
FSB, 3-10, B-7, C-1

Halt Instruction, 3-9, B-6, C-2
HED, 4-20, B-10, C-2
HLT, 3-9, B-6, C-2

IFN, 4-4, B-9, C-2
IFZ, 4-4, B-9, C-2
INA, 3-5, B-4, C-2
INB, 3-5, B-4, C-2
Input
  Data Item Delimiters, F-9
  Floating-Point, F-9
  Free-Field, F-9
  Octal, F-9
  Record Terminator, F-10
Input, Comments Within, F-10
Increment-Skip Instructions, 3-1
Index Register Instructions, 3-5
Indicator, Clear Flag, 2-7
Indirect Addressing, 2-6
Input/Output Instructions, 3-7
Instructions
  Add, 3-1
  Alter-Skip, 3-4
  Bit Processing, 3-3
  Byte Processing, 3-2
  Dynamic Mapping, 3-10, 3-12
  EAU, 3-9
  Extended Arithmetic Unit, 3-9
  Floating Point, 3-10
  Halt, 3-7, 3-9
  Increment-Skip, 3-1
  Index Register, 3-5
  Input/Output, 3-7, 3-8
  I/O, 3-7
  Jump, 3-1
  Load, 3-1
  Logical Operations, 3-2
  Memory Reference, 3-1
  No-Operation, 3-7
  Overflow, 3-7, 3-9
  Register Reference, 3-4
  Shift-Rotate, 3-4
  Store, 3-1
  Word Processing, 3-2
Interrupt Flag, I/O, 2-7
I/O Instructions, 3-7
I/O Interrupt Flag, 2-7
IOR, 3-2, B-2, C-2
ISX, 3-6, B-5, C-2
ISY, 3-6, B-5, C-2
ISZ, 3-1, B-2, C-2

JLY, 3-7, B-5, C-2
JMP, 3-1, B-2, C-2
JPY, 3-7, B-5, C-2
JRS, 3-13, B-7, C-2
JSB, 3-1, B-2, C-2
Jump Instructions, 3-1

Label Field, 2-1
LABEL Symbol, 2-1
LAX, 3-6, B-5, C-2
LAY, 3-6, B-5, C-2
LBT, 3-3, B-3, C-2
LBX, 3-6, B-5, C-2
LBY, 3-6, B-5, C-2
LDA, 3-1, B-2, C-2
LDB, 3-1, B-2, C-2
LDX, 3-6, B-5, C-2
LDY, 3-6, B-5, C-2
Length, Statement, 2-1
LFA, 3-13, B-7, C-2
LFB, 3-13, B-7, C-2
LIA, 3-8, B-6, C-2
LIB, 3-8, B-6, C-2
List Output, 1-6
Listing Control Pseudo Instructions, 4-19
Literals, 2-6
Load Instructions, 3-1
Location Counter, 1-3
Logical Operations, 3-2
LSL, 3-10, B-7, C-2
LSR, 3-10, B-7, C-2
LST, 4-19, B-10, C-2

Map Segmentation, 3-11
MBF, 3-13, B-7, C-2
MBI, 3-13, B-7, C-2
MBT, 3-3, B-3, C-2
MBW, 3-14, B-7, C-2
MEM Violation, 3-12
Memory Reference Instructions, 3-1
MIA, 3-8, B-6, C-2
MIB, 3-8, B-6, C-2
MIC, 4-21, B-10, C-2
MPY, 3-9, B-6, C-2
MVW, 3-2, B-2, C-2
MWF, 3-14, B-7, C-2
MWI, 3-14, B-7, C-2
MWW, 3-14, B-7, C-2

NAM, 4-1, B-9, C-2
No-Operation Instruction, 3-7
NOP, 3-7, B-4, C-2
Numeric Terms, 2-4

Object Program Linkage Pseudo Instructions, 4-5
OCT, 4-17, B-10, C-2
Opcode Field, 2-2
Operand Field, 2-3

Operation Directive, I-2
Operators, Expression, 2-4
Options, Assembly, 1-3
ORB, 4-2, B-9, C-2
ORG, 4-2, B-9, C-2
ORR, 4-2, B-9, C-2
OTA, 3-8, B-6, C-2
OTB, 3-8, B-6, C-2
Output
  Binary, 1-3
  List, 1-6
Overflow Instructions, 3-9

PAA, 3-14, B-8, C-2
PAB, 3-14, B-8, C-2
Paging, 1-1
Passes, 1-1
PBA, 3-15, B-8, C-2
PBB, 3-15, B-8, C-2
Power Fail Characteristics, 3-11
Processing
  Double Defined, I-1
  Literal, I-2
  Pseudo, I-1
  Undefined Label, I-1
  Unused Label, I-2
Program, Source, 1-3
Program Location Counter, 1-3
Program Relocation, 1-1, 1-2
Protected Mode, 3-12
Psuedo Instructions
  Address Definition, 4-11
  Arithmetic Subroutine Calls, 4-20
  Assembler Control, 4-1
  Assembly Listing Control, 4-19
  Constant Definition, 4-14
  Define User Instruction, 4-21
  Linking, 4-5
  Listing Control, 4-19
  Object Program Linkage, 4-5
  Storage Allocation, 4-19
  Symbol Definition, 4-11

RAL, 3-4, B-3, C-2
RAR, 3-4, B-3, C-2
RBL, 3-4, B-3, C-2
RBR, 3-4, B-3, C-2
Register Reference Instructions, 3-4
Registers, Status and Violation, 3-11
Relocatable Expressions, 2-4
Relocation, Program, 1-2
REP, 4-5, B-9, C-2
RPL, 4-10, B-9, C-2
RRL, 3-10, B-7, C-2
RRR, 3-10, B-7, C-2
RSA, 3-15, B-8, C-2
RSB, 3-15, B-8, C-2
RSS, 3-5, B-4, C-2
Running Assemblies Under DOS-III, E-1
RVA, 3-15, B-8, C-2
RVB, 3-15, B-8, C-2

SAX, 3-7, B-5, C-2
SAY, 3-7, B-5, C-2
SBS, 3-3, B-3, C-2
SBT, 3-3, B-3, C-2
SBX, 3-7, B-5, C-2
SBY, 3-7, B-5, C-2
SEZ, 3-4, B-4, C-2
SFB, 3-3, B-3, C-2
SFC, 3-8, B-6, C-2
SFS, 3-8, B-6, C-2
Shift-Rotate Instructions, 3-4
SJP, 3-15, B-8, C-3
SJS, 3-15, B-8, C-3
SKP, 4-20, B-10, C-3
SLA, 3-4, 3-5, B-3, B-4, C-3
SLB, 3-4, 3-5, B-3, B-4, C-3
SOC, 3-9, B-6, C-3
SOS, 3-9, B-6, C-3
Source Program, 1-2, 1-3
SPC, 4-20, B-10, C-3
Specifications
  A and R, F-6
  Conversion, F-4
  E, F-4
  Editing, F-4
  F, F-5
  Format, F-4
  H and " ", F-7
  I, F-5
  O, K, and @, F-6
  X, F-7
  /, F-8
SSA, 3-4, B-4, C-3
SSB, 3-4, B-4, C-3
SSM, 3-15, B-8, C-3
STA, 3-2, B-2, C-3
Statement
  Characteristics, 2-1
  Length, 2-1
STB, 3-2, B-2, C-3
STC, 3-8, B-6, C-3
STF, 3-8, B-6, C-3
STO, 3-9, B-6, C-3
Storage Allocation Pseudo Instruction, 4-19
Store Instructions, 3-1
STX, 3-6, B-5, C-3
STY, 3-6, B-5, C-3
Summary of Instructions, B-1
SUP, 4-19, B-10, C-3
SWP, 3-10, B-7, C-3
SYA, 3-16, B-8, C-3
SYB, 3-16, B-8, C-3
Symbol, Label, 2-1
Symbol Definition Pseudo Instructions, 4-11
Symbols, 1-1
Symbol Table, 1-3
Symbolic Addressing, 1-1
Symbolic Terms, 2-2
SZA, 3-5, B-4, C-3
SZB, 3-5, B-4, C-3

Tape Formats, H-1
Terms
  Numeric, 2-4
  Symbolic, 2-2
  Expression, 2-4

TBS, 3-3, B-3, C-3

UJP, 3-16, B-8, C-3
UJS, 3-16, B-8, C-3
UNL, 4-19, B-10, C-3
UNS, 4-20, B-10, C-3
USA, 3-16, B-8, C-3
USB, 3-16, B-8, C-3

Word Processing Instructions, 3-2

XAX, 3-5, B-5, C-3
XAY, 3-5, B-5, C-3
XBX, 3-5, B-5, C-3
XBY, 3-5, B-5, C-3
XCA, 3-17, B-8, C-3
XCB, 3-17, B-8, C-3
XIF, 4-4, B-9, C-3
XLA, 3-17, B-8, C-3
XLB, 3-17, B-8, C-3
XMA, 3-17, B-8, C-3
XMB, 3-17, B-8, C-3
XMM, 3-18, B-8, C-3
XMS, 3-18, B-8, C-3
XOR, 3-2, B-2, C-3
XSA, 3-18, B-8, C-3
XSB, 3-18, B-8, C-3

PART NO. 92060-90005
REV CODE 1639
Printed in U.S.A. 6/78

HEWLETT hp PACKARD

Sales and service from 172 offices in 65 countries.
11000 Wolfe Road, Cupertino, California 95014