## CONTENTS

### SECTION 1  INTRODUCTION

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>SCOPE OF HANDBOOK</td>
<td>1-1</td>
</tr>
<tr>
<td>REFERENCE DOCUMENTS</td>
<td>1-1</td>
</tr>
<tr>
<td>PRIME 200 ASSEMBLY LANGUAGE</td>
<td>1-2</td>
</tr>
<tr>
<td>Basic Assembly Language Elements</td>
<td>1-3</td>
</tr>
<tr>
<td>Symbolic Instructions</td>
<td>1-6</td>
</tr>
<tr>
<td>Constants, Literals, Variables, and Expressions</td>
<td>1-8</td>
</tr>
<tr>
<td>Symbolic Names</td>
<td>1-9</td>
</tr>
<tr>
<td>Pseudo Operations</td>
<td>1-10</td>
</tr>
<tr>
<td>Macro Facility</td>
<td>1-10</td>
</tr>
<tr>
<td>USING THE MACRO ASSEMBLER</td>
<td>1-12</td>
</tr>
<tr>
<td>Two-Pass Assembly</td>
<td>1-12</td>
</tr>
<tr>
<td>Object Output</td>
<td>1-12</td>
</tr>
<tr>
<td>Listing Format</td>
<td>1-13</td>
</tr>
<tr>
<td>Location Count</td>
<td>1-15</td>
</tr>
<tr>
<td>Symbol Cross Reference Listing</td>
<td>1-15</td>
</tr>
<tr>
<td>ASSEMBLER/LOADER INTERACTION</td>
<td>1-16</td>
</tr>
<tr>
<td>Desectorizing and Address Resolution</td>
<td>1-16</td>
</tr>
<tr>
<td>Extended Addressing Mode</td>
<td>1-17</td>
</tr>
<tr>
<td>Loading Subroutines</td>
<td>1-17</td>
</tr>
<tr>
<td>Memory Map</td>
<td>1-18</td>
</tr>
<tr>
<td>LOADING AND OPERATING PROCEDURES</td>
<td>1-18</td>
</tr>
</tbody>
</table>

### SECTION 2  GENERAL ASSEMBLY LANGUAGE RULES

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>FREE-FORM INPUT TEXT</td>
<td>2-1</td>
</tr>
<tr>
<td>Line Format</td>
<td>2-1</td>
</tr>
<tr>
<td>CONSTANTS, VARIABLES AND EXPRESSIONS</td>
<td>2-5</td>
</tr>
<tr>
<td>Constants</td>
<td>2-5</td>
</tr>
<tr>
<td>Variables</td>
<td>2-7</td>
</tr>
<tr>
<td>Expressions</td>
<td>2-8</td>
</tr>
</tbody>
</table>
SECTION 3  INSTRUCTION STATEMENTS  3-1

INSTRUCTION STATEMENT GENERAL FORMAT  3-1

Label  3-1
Operation Field  3-1
Variable Field  3-7

MEMORY REFERENCE INSTRUCTIONS  3-11

Operation Field  3-13
Variable Field  3-13

INPUT/OUTPUT INSTRUCTIONS  3-14

SHIFT INSTRUCTIONS  3-16

BIT REFERENCE INSTRUCTIONS  3-18

GENERIC INSTRUCTIONS  3-20

SECTION 4  PSEUDO-OPERATIONS  4-1

STATEMENT FORMAT  4-1

ASSEMBLY CONTROLLING PSEUDO-OPERATIONS  4-4

ABS (Set Mode to Absolute)  4-4
REL (Set Mode to Relocatable)  4-4
ORG (Define Origin Location)  4-4
FIN (Insert Literals)  4-6
MOR (More Input Required)  4-6
END (End of Source Statements)  4-6
CF1 Through CF5  4-7
GO, GO TO (Forward Reference)  4-7

LISTING CONTROL PSEUDO-OPERATIONS  4-8

LIST (Enable Listing)  4-8
NLST (Inhibited Listing)  4-8
EJCT (Eject Page)  4-8

LOADER CONTROLLING PSEUDO-OPERATION  4-9

EXD (Enter Extended Addressing Mode)  4-9
LXD (Leave Extended Addressing Mode)  4-9
SETB (Set Base Sector)  4-9
DATA DEFINING PSEUDO-OPERATIONS

DATA (Set Data Constant)  4-11
DEC (Set Decimal Constant)  4-23
DBP (Set Double Precision Constant)  4-24
OCT (Set Octal Constant)  4-25
HEX (Set Hexadecimal Constants)  4-26
VFD (Define Variable Fields)  4-27
BCI (Define ASCII String)  4-28
DAC (Local Address Definition)  4-29
XAC (External Address Definition)  4-30
*** (Dummy Memory Reference Instruction,  4-31

VARIABLE (SYMBOL) DEFINING PSEUDO-OPERATIONS  4-32

EQU (Define Variable),  4-32
SET (Redefine Variable)  4-32

STORAGE ALLOCATION PSEUDO-OPERATIONS  4-34

BSS (Block Starting with Symbol)  4-34
BES (Block Ending with Symbol)  4-34
BSZ (Block Set to Zeroes)  4-34
SETC (Set Common Base Address)  4-35
COMN (Define Common Items)  4-36

PROGRAM LINKING PSEUDO-OPERATIONS  4-38

EXT (Flag External References)  4-38
CALL (External Subroutine Reference)  4-39
SUBR, ENT (Define Entry Points)  4-40

CONDITIONAL ASSEMBLY PSEUDO-OPERATIONS  4-44

IF (Conditional Statement)  4-44
IFM (Continue Assembly if Minus)  4-45
IFP (Continue Assembly if Plus)  4-45
IFZ (Continue Assembly if Zero)  4-45
IFN (Continue Assembly if Not Zero)  4-45
ENDC (End Conditional Assembly Area)  4-46
ELSE (Reverse Conditional Assembly)  4-46
FAIL (Force Error Message)  4-47
SECTION 5      MACRO FACILITY

MACRO DEFINITIONS AND CALLS

MAC (Begin Macro Definition)          5-2
ENDM (End Macro Definition)           5-3
Argument References                  5-3
Macro Calls                           5-3
Arguments Values                      5-4
Argument Substitution                 5-4
Argument Values in Parentheses        5-5
Dummy Words                           5-5
Argument Identifiers                  5-7
Assembler Attribute References        5-8
Local References Within Macros        5-8

MACRO LISTING AND ASSEMBLY CONTROL    5-10

LSTM (List Macro Expansions)          5-10
LSMD (List Macro Expansions - Data Only)  5-11
NLSM (No Listing of Macro Expansions)  5-12
BACK, BACK TO (Loop Back - Macros Only) 5-13
SAY (List Message to Operator)        5-13

MACRO EXAMPLES                        5-15

SECTION 6      SOURCE FILE MERGING COMMANDS    6-1

$INSERT       6-1
$UPDATE        6-2
$COPY          6-2
$DONE          6-2

APPENDICES

A  PRIME 200 Instructions (Op Code Order)  A-1
B  PRIME 200 Instructions (Class Order)    B-1
C  PRIME 200 Instructions (Mnemonic Order) C-1
D  I/O Device Codes                       D-1
E  ASCII Character Codes                  E-1
F  Object File Formats                    F-1
G  Assembler Error Messages               G-1
ILLUSTRATIONS

<table>
<thead>
<tr>
<th>Figure</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1-1</td>
<td>Example of PRIME 200 Macro Assembly Language Statements</td>
<td>1-5</td>
</tr>
<tr>
<td>1-2</td>
<td>Interpretation of Symbolic Instruction</td>
<td>1-7</td>
</tr>
<tr>
<td>1-3</td>
<td>Example of Assembly Listing</td>
<td>1-14</td>
</tr>
<tr>
<td>1-4</td>
<td>Example of Memory Map</td>
<td>1-19</td>
</tr>
<tr>
<td>2-1</td>
<td>Source Input Line Formats</td>
<td>2-2</td>
</tr>
<tr>
<td>3-1</td>
<td>General Format of Instruction Statements</td>
<td>3-8</td>
</tr>
<tr>
<td>3-2</td>
<td>Assembly and Loading of Memory Reference Instruction</td>
<td>3-12</td>
</tr>
<tr>
<td>3-3</td>
<td>Assembly and Loading of Input/Output Instruction</td>
<td>3-15</td>
</tr>
<tr>
<td>3-4</td>
<td>Assembly and Loading of Shift Instructions</td>
<td>3-17</td>
</tr>
<tr>
<td>3-5</td>
<td>Assembly and Loading of Bit Reference Instructions</td>
<td>3-19</td>
</tr>
<tr>
<td>3-6</td>
<td>Assembly and Loading of Generic Instructions</td>
<td>3-21</td>
</tr>
<tr>
<td>4-1</td>
<td>General Format of Pseudo-Operation Statements</td>
<td>4-3</td>
</tr>
<tr>
<td>4-2</td>
<td>Single Precision Fixed Point Constants</td>
<td>4-14</td>
</tr>
<tr>
<td>4-3</td>
<td>Double Precision Fixed Point Constants</td>
<td>4-17</td>
</tr>
<tr>
<td>4-4</td>
<td>Floating Point Word Formats</td>
<td>4-19</td>
</tr>
</tbody>
</table>

TABLES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3-1</td>
<td>Summary of PRIME 200 Instruction Codes</td>
<td>3-2</td>
</tr>
<tr>
<td>4-1</td>
<td>Summary of Pseudo-Operations</td>
<td>4-2</td>
</tr>
<tr>
<td>4-2</td>
<td>Numerical Formats in DATA Statements</td>
<td>4-13</td>
</tr>
</tbody>
</table>
SECTION 1
INTRODUCTION

SCOPE OF HANDBOOK

This handbook is a detailed reference manual for the PRIME 200 Macro Assembly Language. It is organized in six sections for ease of reference.

This section introduces the assembly language, describes the action of the program, and discusses the interaction of the assembler with its companion program, the PRIME 200 Linking Loader.

Section 2 discusses statement formats and language features common to all types of assembly language statements.

Section 3 contains the rules for forming instruction statements using PRIME 200 instruction mnemonics.

Section 4 describes pseudo-operations (directives to the assembler and loader).

Section 5 defines the Macro facility, a way to define program statements that can be called for execution by easily interpreted English language statements.

Section 6 defines commands used to invoke functions of RDALN, routine that merges lines from two or more source files during assembly.

The handbook is concluded by several appendices and a detailed subject index.

REFERENCE DOCUMENTS

The following publications are recommended to supplement this handbook:

PRIME 200 Programmer's Reference Manual
PRIME 200 Operator's Guide
PRIME 200 DOS Reference Manual
PRIME 200 RTOS Reference Manual
The PRIME 200 Macro Assembly Language has the usual provisions for symbolic instructions, symbolic addresses, and control pseudo-operations expected by computer users. It also offers many other advanced features:

* **Free Format:** Source statements are independent of column boundaries and permit free use of spaces. Multiple statements per line are permitted, and statements may be continued from line to line.

* **Symbols:** Symbols or Variables assigned to address and data locations may contain up to 32 characters.

* **Constants:** Wide variety of constant forms: decimal, octal, hexadecimal, ASCII, double precision, floating point, literals.

* **Expressions:** Symbols and constants may be linked in expressions using 14 different arithmetic, logical, and shift operators.

* **Pseudo-Operations:** Over 50 pseudo-operations for assembly control, listing control, loader control, data definition, variable definition, storage allocation, program linking, and conditional assembly.
* Macro Facility: Programmer can define macros to be called by application-related language statements. Arguments are identified by position or flagged by key words. Looping, local references, and nesting are permitted.

The main purpose of an assembly language is to reduce the clerical chores required to prepare a binary program that can be executed by the computer. Of course, it is possible to look up the binary code for a given instruction and key it into a memory location using front panel controls. For example, an instruction to load the A register from location '377 of sector 0 would be the octal code '004377. The octal code for any PRIME 200 instruction can be determined from the Programmer's Reference Manual. (Also see Table 3-1.)

But manual key-in of programs is tedious, error prone, and the bare binary codes can only be interpreted by a painstaking analysis. This mode of program entry is usually limited to key-in loader bootstrap programs and short test sequences. A symbolic assembly language has become the universal means of preparing programs of any size. An assembly language provides a vocabulary of symbolic, or mnemonic, codes - and a grammar of statement forms - to represent machine language instructions in a format that is easily read and interpreted by the original programmer or any other reader familiar with the language.

Basic Assembly Language Elements

Figure 1-1 illustrates a section of a typical program written in PRIME 200 Macro Assembly Language. The basic unit of information processed by the PRIME 200 assembler is the line. When originated at an ASR-33 Teletype keyboard, a line consists of up to 72 ASCII characters (75 for ASR-35) occurring between carriage return - line feed characters. When input is from unit record equipment, a line consists of an 80-character card field.

There are statement lines, comment lines and change page heading lines. A statement line has a space in column 1, an optional label, one or more statements, and an optional comment field. A comment line has an asterisk in column 1; the rest of the line is ignored except for listing purposes.
A change page heading line has an apostrophe in column 1; the rest of the line becomes the page heading for all subsequent listing pages.

A label is an ASCII character string that identifies the locations count of the first statement in the line. Examples are the name of the entry point for a subroutine or the symbolic name of a storage location. These and other common features of the language are described in detail in Section 2.

The PRIME 200 processes four types of statements, each with a unique format: instructions, pseudo-operations, macro calls, and file merging commands. These are described in detail in Sections 3 through 6, respectively.
*SHORT DEMO PROGRAM EXAMPLE
*
* 
ABS
ORG 13000
CRA

COTA 1770
SR1
JMP* DOS

R
IRX
JMP A

R
JRX
JMP B

AOA
FAIL
JMP C

DOS
END

DISPLAY A REG IN PANEL IND
DATA INDICATORS COUNT SLOWLY
DELIBERATE ERROR LINE

Figure 1-1. Example of PRIME 200 Macro Assembly Language Statements
Symbolic Instructions

The A Register load operation mentioned above can be represented in the form:

LDA '377

where LDA is the mnemonic of the LDA instruction and '377 is the octal representation of the memory address where the data is located. The programmer may also control the flag and tag bits symbolically, as in:

LDA* '377,1

where the asterisk specifies indirect addressing and the ,1 specifies indexing. After being processed by the assembler and loader, this statement would be converted into a binary instruction word, shown in Figure 1-2. The resulting word would have an octal value of '144377.
Figure 1-2. Interpretation of Symbolic Instruction.
Constants, Literals, Variables, and Expressions

This assembler permits a variety of forms for data constants, thereby eliminating conversions from decimal to binary, octal, or hexadecimal. Examples:

\[
\begin{align*}
'123 & \quad -'777 & \quad '-1777 & \quad \text{Octal} \\
$89AB & \quad -$FFFF & \quad $-00FF & \quad \text{Hexadecimal} \\
1234 & \quad -9999 & \quad 32767 & \quad \text{Decimal}
\end{align*}
\]

The constant forms shown above are all single-precision (i.e. are converted to a single 16-bit data word - 15 magnitude bits plus sign). For decimal numbers, double precision and floating point quantities may be specified:

\[
\begin{align*}
1.23BB6 & \quad 1.23EE2BB6 & \quad \text{Fixed Point Double Precision} \\
1.23E2 & \quad 1.1092 & \quad \text{Floating Point Single Precision} \\
1.23EE2 & \quad \quad & \quad \text{Floating Point Double Precision}
\end{align*}
\]

The assembler also accepts ASCII constants:

\[
\begin{align*}
C'A' & \quad (\text{The letter A, left justified in a 16-bit word}) \\
C'AB' & \quad (\text{The letters A and B packed into a 16-bit word.})
\end{align*}
\]
Another form of constant with a self-defining symbolic name is the **literal**:

- '77 Octal
- $39FB Hexadecimal
- 199 Decimal
- C'X' (one character) ASCII
- C'XY' (two characters)

Variables, also called symbols or symbolic names, may be assigned to identify memory locations. Symbols are defined by being used in the label field of a statement, or by the EQU or SET pseudo-operations. The assembler accepts alphanumeric symbols of up to 32 characters:

- A
- ALPHA
- ABCDEFGHIJKLMNOPQRSTUVWXYZ123456

Expressions may be formed using constants or variables, linked by 14 different arithmetic, logical and shift operators:

- A + 3 ALPA*(4 - B) A .LS.(ALPA/5)
- A .AND. '3737 BETA .GE.A+$FF

**Symbolic Names**

Symbolic names may be coined by the programmer and assigned to memory locations, so that data locations and program entry points can be specified by self-explanatory codes rather than numerical values. For example, the load A instruction could be coded as

- LDA* DATA,1

provided the symbol DATA is defined somewhere in the program as equal to memory location '377. During the first pass of an assembly operation, the assembler builds a
symbol table that relates each symbol (also called variable, or symbolic name) to the location where it is defined. On the second pass, the numerical value of each symbol is substituted for the alphanumeric expression, wherever it is used in an address field.

Symbolic names can, in many cases, be modified or processed by arithmetic operators, as in

```
LDA DATA-1
LDA DATA*4-1
```

**Pseudo Operations**

In addition to instruction statements, the assembly language provides pseudo operation statements that give the programmer control of the assembly process itself and of the loading operation that follows assembly.

In the example of Figure 1-1, several pseudo-operations are used. The program example begins with an ABS, specifying absolute loading mode. An ORG statement sets the assembler's location count (discussed later) to '3000. A OCT statement equates the symbol DOS to the octal quantity '30000. The program example ends with a mandatory END statement. These and many other pseudo-operations are described in detail in Section 4.

**Macro Facility**

The macro feature of this assembler enables the programmer to define functions that can be expressed in easily interpreted English (or other) language statements:

```
TRANSFER DATA TO DAC
TURN ON VALVE 312
```

Statements of this sort are made possible by a process called macro definition. With the aid of the MAC and ENDM pseudo-operations, a system programmer can create macro prototypes.
The TRANSFER statement, above, might be defined by the following sequence of statements:

```
TRANSFER MAC TO
LDA <1>
OTA <2>
ENDM
```

The MAC pseudo-operation introduces the macro definition by assigning the name TRANSFER to the macro and identifying the word TO as a dummy word (a word that can be used to increase the intelligibility of macro calls without being mistaken for an argument).

Variable fields of the LDA and OTA instructions call for arguments, symbolized by numerals enclosed within angle brackets. Values for arguments are supplied by the macro call statements. For example the statement TRANSFER DATA to DAC calls for the TRANSFER macro to be assembled, with the symbol DATA substituted for argument <1>, and the symbol DAC substituted for argument <2>. The TRANSFER macro would then be assembled as follows:

```
```

After a set of macros has been defined by a system-level programmer, a specialist in a particular application field can formulate macro calls in plain language to solve his application problems, without becoming involved in the details of assembly language programming. Definition, listing, and assembly are discussed in detail in Section 5.
USING THE MACRO ASSEMBLER

The Macro Assembler translates ASCII source files and produces an object file, for processing by the Linking Loader, and an optional listing file, to be printed as a record of the source language statements and the octal codes to which they have been translated. The files may be printed or punched on tape during assembly or they may simply be stored (on disk, for example) until they are needed. Device options are specified by register settings at the start of assembly.

Two-Pass Assembly

The assembler program itself is first loaded into computer memory. The assembler occupies approximately 4K memory locations; the absolute location in CPU memory depends on the amount of memory available, and the type of system (DOS-based, stand-alone, etc.).

To use the assembler, the operator sets up an input device containing a source program file. Devices to receive the object files and listing output (optional) are specified by entries in CPU registers and the assembler is started.

This two-pass assembler first reads the source file to locate and assign values to any alphanumeric variables (symbols) used in instruction or pseudo-operation variable fields. The source file is then returned to the beginning and read again. On the second pass, the assembler substitutes numerical values for all variables and evaluates expressions, thus converting symbolic references to 16-bit binary quantities. The assembler then outputs the object file and a listing, if requested.

Object Output

The object output of the assembler is in a special format suitable for input to the Linking Loader. Instructions, data constants, and directives to the Loader are encoded as blocks of data in various sizes and formats. (For details, see Appendix F.) When object files are punched on paper tape, they are in an "invisible" character format; none of the frames punched on the tape will cause printing on an ASR. (This saves paper by eliminating nonsense printout when the ASR is used as the loading device.)
Listing Format

The object file is in an arbitrary binary format that is meaningful only to the loader, but the optional listing file pairs an octal representation of the object code with the actual source input statements they represent, in a format that is meaningful to the programmer.

Figure 1-3 shows a section of a typical assembly listing and defines the main features. The format is organized in columns, but when long labels or other free format features are encountered, extra space is used as required.

Each page of the listing begins with a header provided by the source statements, and a sequential page number. The first statement in a program is used as the initial page header, unless it starts with a quotation mark ("'). If column 1 of any statement contains an apostrophe ('), columns 2-72 of that statement become the title for all pages that follow until a new title is specified.

Columns 1-4 are reserved for error flags. (See Appendix G.) Columns 5-9 contain an octal assignment address location count and columns 11-19 contain the octal object code generated by each statement. Columns 21-26 contain a decimal line sequence number and columns 28-108 contain the source statement (ASCII Image) truncated if necessary depending on printer limitations.
Figure 1-3: Example of Assembly Listing
Location Count

The assembler assigns a sequential location count to each element in the object code that will be converted to a CPU memory location. (Instruction statements always generate one line of code; data defining pseudo-operations may generate one or more lines, depending on the constant format.)

The starting value for the location count is zero, unless another origin is specified by the ORG pseudo-operation. The assembler normally increments the location count by 1 after each entry but a new count can be established by another ORG statement at any point in a program. In the example of Figure 1-3, an ORG statement sets the origin to '3000 and the location count is stepped sequentially from that value.

Figure 1-3 also shows how symbols are assigned numerical values in relation to location counts. The symbol A for example, is equated to '3000 when it is used in the label field of the IRS instruction in that location.

The address field of the JMP instruction in location 3005 contains a reference to the symbol A. Notice that the JMP instruction is assembled with the assigned value of A ('3004) in its address field.

Symbol Cross Reference Listing

At the end of the assembly listing appears a cross-reference listing of each symbol's name (in alphabetical order), the symbol's assignment address or value, and a list of all reference to that symbol. Each reference is identified by a 4-digit decimal line number.

The information necessary for the cross-reference listing is stored in the symbol table. If, during assembly, the symbol table becomes full, cross-reference information is sacrificed in order for assembly to continue. If this occurs, the cross-reference listing will contain only the alphabetic symbol names and their assignment addresses.

If listing is inhibited (by NLST pseudo-operation), the cross reference listing is not listed. The same listing device is used for the cross-reference as is used for the Pass 2 assembly listing.

The last line of the listing specifies the version of the assembler and the number of lines containing error flags.
ASSEMBLER/Loader Interaction

The Linking Loader is required to interpret the object code blocks, form 16-bit binary instruction and data codes, and load them in the proper locations of main memory. The actual location in which a word is loaded depends on whether absolute (ABS) or relocatable (REL) mode has been specified. (ABS is the assembler's default mode.) In absolute mode, the assembler-assigned location count becomes the actual instruction location. In relocatable mode, an address offset, entered into a register at the start mode of loading, is added to the location count. The E32R and E64R addressing modes further modify this procedure.

Two or more relocatable programs can be packed together anywhere in memory without wasted space, even though the final locations are unknown at the time of programming. The "linking" feature of the assembler-loader combination permits main programs and subroutines to share common data locations and entry points.

Desectorizing and Address Resolution

In assembly language there is no way to specify that the sector bit of memory reference instructions is to be set, except to count instructions and data locations and deliberately keep track of the current sector. In a program of any length, bookkeeping of this type would become tedious. Instead, the assembler and loader take over this function. They jointly keep track of sector information and set or clear the sector bit of each memory instruction at the time it is loaded.

The binary object code output of the assembler includes a 14-bit or 15-bit address for each memory reference instruction, depending on whether or not extended addressing is in effect. (See EXD pseudo-operation).

In 16-sector addressing mode (E16S), the assembler presents a 14-bit binary address to the loader, along with an indirect bit and indexing bit. As the loader processes the instruction, it compares the instruction's 14-bit address with the current location count. If the instruction and the address are in the same sector, the loader truncates the address to 9 bits, loads it into the instruction address field (bits 8-16) and sets the instruction's sector bit (bit 7).
However, if the instruction's 14-bit address specifies a different sector than the one containing the instruction, the loader assigns a location in a table of cross-sector indirect words and loads the 14-bit address (plus indirect and indexing bits) in that location. The indirect bit and address field of the instruction word itself are set to point to the indirect word. Since the indexing bit is moved to the indirect location, the index bit of the instruction itself is cleared.

Ordinarily, the table of indirect words begins at location '100 of Sector zero and grows upward as required. However, another base sector can be specified by the SETB pseudo-operation, and the starting location for the links can be altered by a register setting at the time of loading.

Extended Addressing Mode

If extended addressing mode has been set up by the EXD pseudo-operation, the assembler presents 15-bit addresses to the loader. (Bit 2 of the address is interpreted as a magnitude bit rather than the index bit.) The check for cross-sector references is made as usual, and an indirect link is formed if necessary. The full 15-bit address is stored in a resulting indirect link location (indexing cannot be specified).

It is important to specify that code be loaded in the mode in which it is to execute. If the source program contains a EXD pseudo-operation, extended addressing mode must also be set up for the CPU by an E32S or E32R instruction.

Loading Subroutines

If the main program calls for external subroutines, the loader halts and waits for the user to assign the library or other files containing subroutines to the input device already selected. The loader then identifies and loads every subroutine called by the main program. Subroutines are desectorized and linked together in consecutive memory locations unless new ORG values are assigned.
Memory Map

At any time during loading of a series of programs and subroutines, the loader can be directed to print a memory map. The map shows the locations occupied by the program in memory, specifies locations for common storage, shows subroutine entry locations, and identifies subroutines that have been called but are not yet loaded. A memory map for the program example discussed earlier appears in Figure 1-4.

LOADING AND OPERATING PROCEDURES

Loading and operating procedures for the Macro Assembler vary according to the type of installation, memory size, and supporting software. Appropriate procedures are available in one of the following documents:

PRIME 200 Operator's Guide
PRIME 200 DOS Operator's Guide
Figure 1-4. Example of Memory Map
SECTION 2
GENERAL ASSEMBLY LANGUAGE RULES

The following language features are common to all types of statements. Features that are peculiar to instruction statements, pseudo-operation statements, and macro calls are defined in later Sections.

FREE-FORM INPUT TEXT

Input text for the PRIME 200 assembler may be prepared in a number of ways. Perhaps the most common is text prepared at a teleprinter with the aid of the PRIME 200 text editor. The resulting source program exists as a file in memory or on the disk, and can be punched on paper tape in ASCII format. For text prepared in this way, the basic unit of information is a line, as delimited by carriage return-line feed (CRLF) characters. Because of the mechanical limitations of most teleprinter devices, lines are usually limited to 72 or 75 characters.

Input text may also be coded by hand on paper forms which are then keypunched to produce unit record cards. Each card is equivalent to one teleprinter line, but may contain up to 80 characters.

Line Format

PRIME-200 assembler input lines consist of labels, statements, and comments strung together in a free (column independent) format without regard for tabulation positions or arbitrary column boundaries. The assembler recognizes statements within a line, and subfields within statements, by delimiters consisting of spaces, colons, commas, backslash (tab character), and semicolons. (Labels and comments are optional.) For examples see Figure 2-1.

Labels: Labels are used to assign mnemonic codes to memory locations - for example, the name of a subroutine entry location or the symbolic address of a constant or storage cell. Labels are optional. If a line includes a label, the first character of the label must be in column 1 of the line. (Otherwise column 1 must be blank.) Labels must conform to the character set and size prescribed for variables.
Figure 2-1. Source Input Line Formats
Statements: The PRIME 200 assembler accepts four types of statements: symbolic instructions, pseudo-operations, macro calls and commands to the RDALN source file update program. Each of these has a different sub-field format, and is described in a later section of this manual.

If the line does not have a label, the first statement begins with column 2 or the first non-space character. Otherwise the statement begins with the first non-space character following the label.

A statement is terminated by two spaces or column 73, whichever comes first. Subsequent characters are assumed to be comments and are ignored except for listing.

Multiple Statements per Line: Statements can be packed two or more per line, separated by colons (:). The first non-space character following the colon is processed as the first character of the next statement. The last statement in the line is terminated by two spaces or column 73, and the rest of the line treated as comments.

If the line begins with a label, the label is attached to the first statement during assembly.

Continued Statements: The last statement in a line may be interrupted by a semicolon (;) and continued on the next line. The rest of the line following the semicolon is treated as comments. Processing of the statement continues with the first non-space character in the following line. Semicolons occurring within comments are not interpreted as a continuation request.

Comments: All text following column 72, or following a statement and two or more spaces, is treated as comments, and ignored except for listing. Comments can contain all printing ASCII characters.

Comment Lines: If column 1 of a line contains an asterisk (*) the rest of the line is treated as comments. (Comment lines can be used to continue comments begun in the preceding statement line.)

Change Page Heading Lines: A line that contains an apostrophe (') in column 1 is assumed to contain the text of all subsequent page titles.
Examples: The following assembly listing illustrates many of the free-form input features.

```
01252: 02 01215 0198 ABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789 LDA ALPA
01253: 02 01257 0199 LDA ABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789
0204) *
0205) *-----TEST OF FREE FORMAT AND LINE CONTINUATION
0206) AB,
0207) C   A,
01254: 06 01215 0204 DD ALPA.
01255: 000001 0205
01256: 01 01215 0206 AB JMP ALPA
01257: 140114 0207 IRX
01260: 041666 0208 LLS 10
01261: 26 01220 0209 ADD BETA.1
01262: 041667 0210 LLL 9 COMEENT
01263: 02 000001 0211 *
01264: 02 0213 ABX LDA 1: ADD 2: STA 3
01267: 000281
01270: 140040
```

2-4
CONSTANTS, VARIABLES AND EXPRESSIONS

Constants

Symbolic names and address expressions used within the assembler program statements may contain decimal, octal, hexadecimal or ASCII constants.

Magnitudes: Numerical constants used in expressions are limited to the magnitudes that can be represented by the single-precision (16-bit) PRIME 200 binary arithmetic word:

<table>
<thead>
<tr>
<th>Type</th>
<th>Max. Negative</th>
<th>Max. Positive</th>
</tr>
</thead>
<tbody>
<tr>
<td>Decimal</td>
<td>-32768</td>
<td>+32767</td>
</tr>
<tr>
<td>Octal</td>
<td>-'100000</td>
<td>+'777777</td>
</tr>
<tr>
<td>Hexadecimal</td>
<td>-$4000</td>
<td>+$7FFF</td>
</tr>
</tbody>
</table>

Leading zeroes can be omitted. If the sign is omitted, the quantity is assumed to be positive.

Double precision and floating point constants may be set up using the data defining pseudo-operations described in Section 4. However, these constants cannot be used within expressions.

Decimal Constants: All numerical quantities are assumed to be decimal (base 10) unless they are tagged with the octal, hexadecimal or ASCII designator symbols shown below.

Octal Constants: Octal constants (base 8) are identified by an apostrophe or O designator:

'123 or O'123' or O'123
'+123 or O'+123' or O'+123
'-123 or O-123' or O'-123

Note that the sign follows the octal designator. In expressions, however, the minus operator must precede the designator: SYMBOL-'123 is legal, but SYMBOL+-'123 is not.

Hexadecimal Constants: Hexadecimal constants (base 16) are identified by a dollar sign or X designation:

$30BF or X'30BF'
$-30BF or X'-30BF'
Here also the sign follows the designator, but in expressions the minus operator must precede the designator: SYMBOL-$30BF is legal, but SYMBOL+$-30BF is not.

The hexadecimal digit values are:

<table>
<thead>
<tr>
<th>Hexadecimal Digit</th>
<th>Decimal Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-9</td>
<td>0-9</td>
</tr>
<tr>
<td>A</td>
<td>10</td>
</tr>
<tr>
<td>B</td>
<td>11</td>
</tr>
<tr>
<td>C</td>
<td>12</td>
</tr>
<tr>
<td>D</td>
<td>13</td>
</tr>
<tr>
<td>E</td>
<td>14</td>
</tr>
<tr>
<td>F</td>
<td>15</td>
</tr>
</tbody>
</table>

ASCII Constants: One or two eight-bit ASCII character codes can be represented by the following notation:

C'A' Represents the ASCII code for the character A, left-justified in a 16-bit field with a trailing space character.

C'AZ' Represents the codes for the ASCII characters A and Z, packed into a 16-bit field with A left justified and Z occupying the rightmost 8 bits.

Any printing character of the ASCII character set can be used.

Examples:
Variables

Variables are alphanumeric strings, often called "symbols" or "symbolic names", that are equated to numerical values in various ways. If a variable is used as the label of a statement, it is assigned the value of the location count for that statement. Variables may also be defined by the SET, EQU and DAS pseudo-operations described in Section 4.

Variables can be from 1 to 32 characters long. The first character must be a letter (A-Z), and the remaining characters may be letters, numerals (0-9) or the dollar sign ($). Variables containing more than 32 characters are allowed but only the first 32 characters are recognized by the assembler. Variable names must be unique (cannot be defined more than once).

Examples: The following examples show some of the ways a variable "VARI35$" can be used.

```
00002: 02 01234 (0006) P2#7
00003: 02 01333 (0007)
       001234 (0008) VARI35$
00004: 01 01232 (0009)
       JMP VARI35$-2
```

LDA VARI35$
LDA VARI35$+77
SET 1234
Expressions

Expressions consist of constants or variables joined by operators. All variables within an expression must be defined as single precision values or addresses. Absolute, relative and external values cannot be used in the same expression.

Examples:

```
LDA K100+3
STA ++VAR1554-K100
```

Operators: The PRIME 200 assembler is able to process the following arithmetic, logical, relational and shift operators while evaluating expressions in instruction address fields, arguments in macro calls, etc.

* Arithmetic Multiply
/ Arithmetic Divide
+ Arithmetic Add
- Arithmetic Subtract
.OR. Logical OR (16 bits)
.XOR. Logical XOR (16 bits)
.AND. Logical AND (16 bits)
.EQ. Relational EQ (resulting in 0 or 1)
.NE. Relational NE (resulting in 0 or 1)
.GT. Relational GT (resulting in 0 or 1)
.LT. Relational LT
.GE. Relational GE (resulting in 0 or 1)
.LE. Relational LE (resulting in 0 or 1)
.RS. Logical Right Shift (16 bits)
.LS. Logical Left Shift (16 bits)
Space Conventions: Operators may be followed by a single space (optional). The logical, relational and shift operators must be preceded with a space so that the period beginning these symbols will not be interpreted as a decimal point.

Operator Priority: In expressions with more than one operator, the order of evaluation is governed by operator priority. The operator with the highest priority is performed first. In cases of equal priority, the evaluation proceeds from left to right. Parentheses may be used to alter the natural order of evaluation.

<table>
<thead>
<tr>
<th>Priority</th>
<th>Operator(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Highest</td>
<td>* /</td>
</tr>
<tr>
<td></td>
<td>+ -</td>
</tr>
<tr>
<td></td>
<td>.RS .LS .GT .GE .EQ .NE .LE .LT .AND .OR .XOR</td>
</tr>
<tr>
<td>Lowest</td>
<td></td>
</tr>
</tbody>
</table>

Relational Operators: These are most often used in the argument field of IF pseudo-operations. However they may be used in other expressions. Examples of the correct syntax are:

```
01304:  000001 (0234)  ------TEST RELATIONAL OPERATORS
01305:  000000
01306:  000000
01307:  000000 (0236)
01308:  000000
01309:  000000
01310:  000000
01311:  000000
01312:  000000 (0237)
01313:  000000
01314:  000000
01315:  000000 (0238)
01316:  000000
01317:  000000
01318:  000000 (0239)
01319:  000000
01320:  000000
01321:  000000
01322:  000000
01323:  000000 (0240)
01324:  000000
01325:  000000
```

DATA 5 EQ 5, 5 .EQ 6, 6 .EQ 5
DATA 5 NE 5, 5 NE 6, 6 NE 5
DATA 5 GT 5, 5 GT 6, 6 GT 5
DATA 5 LT 5, 5 LT 6, 6 LT 5
DATA 5 .GE 5, 5 .GE 6, 6 .GE 5
DATA 5 .LE 5, 5 .LE 6, 6 .LE 5
Shift Operators: The shift operators perform a logical right or left shift of an expression, using the syntax:

\[
\begin{bmatrix}
\text{Argument Expression} & .LS. & \text{Shift Count Expression} & .RS.
\end{bmatrix}
\]

where the shift count expression has a numerical value from 1 to 16.

Examples:

\begin{verbatim}
(0241) *
(0242) *-----TEST SHIFT OPERATOR:
01326: 000101 (0243) DATA '1010 .RS. 3, '1010 .LS. 3, '1010 LS. 4
01327: 010100
01328: 020200
01329: 010100 (0244) DATA '1010 .RS. -3, '1010 .LS. -3
01330: 000101
(0245) *
\end{verbatim}
**Logical Operators:**

Examples are shown below.

| 01301: 001100 (0230) | DATA 1100 .OR. '0101 |
| 01302: 001001 (0231) | DATA 1100 .XOR. '0101 |
| 01303: 000100 (0232) | DATA 1100 .AND. '0101 |
| (0233) * |
Sign Conventions: In expressions containing the + and - operators, integer constants may be signed:

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>02007</td>
<td>02.01715 (0013)</td>
<td>P2#$12 LDA ZILCH-#3E</td>
</tr>
<tr>
<td>02010</td>
<td>02.01715 (0014)</td>
<td>LDA ZILCH+#-3E</td>
</tr>
<tr>
<td>02011</td>
<td>02.01755 (0015)</td>
<td>LDA ALPHA-#37</td>
</tr>
<tr>
<td>02012</td>
<td>02.01755 (0016)</td>
<td>LDA ALPHA+#-37</td>
</tr>
<tr>
<td>02013</td>
<td>0000006 (0017)</td>
<td>ZILCH DEC 0</td>
</tr>
<tr>
<td>02014</td>
<td>000147 (0018)</td>
<td>ALPHA DATA 99</td>
</tr>
</tbody>
</table>
SECTION 3

INSTRUCTION STATEMENTS

This section defines the form of all PRIME 200 instruction statements, shows how instructions are processed by the assembler and loader, and covers syntax elements peculiar to instruction statements.

Table 3-1 lists the instruction mnemonics acceptable to the assembler. Note that some instructions have two mnemonics; those in parentheses are accepted for compatibility with other assemblers. Mnemonics in Table 3-1 are in order by functional type. The instructions are sorted by op-code in Appendix A, by class in Appendix B, and by mnemonic in Appendix C.

INSTRUCTION STATEMENT GENERAL FORMAT

The essential elements of an instruction statement are an operation field and a variable field, separated by spaces, a comma, or a backslash tab symbol (\), as shown in Figure 3-1. The content of each field depends on the type of instruction being processed. Memory reference instructions have different requirements than I/O, shift, bit reference, or generic instructions. Label and comment fields are optional.

Label

If a label is present, it is assigned the current location count and entered in the symbol table.

Operation Field

The operation field must contain one of the PRIME 200 instruction mnemonics listed in Table 3-1. An asterisk, for indirect addressing, applies to memory reference instructions only. Parentheses are ignored:

<table>
<thead>
<tr>
<th>Code</th>
<th>Operation</th>
<th>Mnemonic</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>01242</td>
<td>02 00012</td>
<td>(0181)</td>
<td>(LDA)</td>
</tr>
<tr>
<td>01243</td>
<td>42 00112</td>
<td>(0182)</td>
<td>(LDA*)</td>
</tr>
<tr>
<td>01244</td>
<td>42 00112</td>
<td>(0183)</td>
<td>(LDR)</td>
</tr>
<tr>
<td>01245</td>
<td>62 00112</td>
<td>(0184)</td>
<td>(LDA)*</td>
</tr>
</tbody>
</table>

10
10
10
10.1
Table 3-1. Summary of PRIME 200 Instruction Codes

<table>
<thead>
<tr>
<th>CLASS</th>
<th>OP CODE</th>
<th>MNEMONIC</th>
<th>DEFINITION</th>
</tr>
</thead>
<tbody>
<tr>
<td>G</td>
<td>140040</td>
<td>CRA</td>
<td>CLEAR A</td>
</tr>
<tr>
<td>G</td>
<td>140014</td>
<td>CRB</td>
<td>CLEAR B</td>
</tr>
<tr>
<td>G</td>
<td>140016</td>
<td>CRL</td>
<td>CLEAR LONG (A AND B)</td>
</tr>
<tr>
<td>MR</td>
<td>02</td>
<td>LDA</td>
<td>LOAD A</td>
</tr>
<tr>
<td>MR</td>
<td>04</td>
<td>STA</td>
<td>STORE A</td>
</tr>
<tr>
<td>MR</td>
<td>15</td>
<td>LDX</td>
<td>LOAD INDEX (ASSEMBLER SETS INDEX BIT)</td>
</tr>
<tr>
<td>MR</td>
<td>15</td>
<td>STX</td>
<td>STORE INDEX (ASSEMBLER CLEARS INDEX BIT)</td>
</tr>
<tr>
<td>MR</td>
<td>13</td>
<td>IMA</td>
<td>INTERCHANGE MEMORY AND A</td>
</tr>
<tr>
<td>G</td>
<td>000201</td>
<td>IAB</td>
<td>INTERCHANGE A AND B</td>
</tr>
<tr>
<td>G</td>
<td>140104</td>
<td>XCA</td>
<td>TRANSFER A TO B AND CLEAR A</td>
</tr>
<tr>
<td>G</td>
<td>140204</td>
<td>XCB</td>
<td>TRANSFER B TO A AND CLEAR B</td>
</tr>
<tr>
<td>G</td>
<td>000111</td>
<td>CFA</td>
<td>COMPUTE EFFECTIVE ADDRESS</td>
</tr>
</tbody>
</table>

ARITHMETIC

| MR    | 06      | ADD      | ADD MEMORY TO A |
| MR    | 07      | SUB      | SUBTRACT MEMORY FROM A |
| G     | 141206  | AAO (A1A) | ADD ONE TO A |
| G     | 140304  | A2A      | ADD TWO TO A |
| G     | 140110  | SAA (S1A) | SUBTRACT ONE FROM A |
| G     | 140310  | S2A      | SUBTRACT TWO FROM A |
| G     | 141216  | ACA      | ADD C-BIT TO A |
| G     | 140320  | CSA      | COPY SIGN TO C-BIT. SET SIGN OF A PLUS |
| G     | 140100  | SSP      | SET SIGN OF A PLUS |
| G     | 140500  | SSM      | SET SIGN OF A MINUS |
| G     | 140024  | CHS      | CHANGE SIGN OF A |
| G     | 140407  | TCA      | TWO'S COMPLEMENT A |
| G     | 000205  | PIM      | POSITION FOR INTEGER MULTIPLY |
| G     | 000211  | PID      | POSITION FOR INTEGER DIVIDE |
| MR    | 16      | MPY      | MULTIPLY |
| MR    | 17      | DIV      | DIVIDE |
| MR | 02 | DLD | DOUBLE PRECISION LOAD |
| MR | 04 | DST | DOUBLE PRECISION STORE |
| MR | 06 | DAD | DOUBLE PRECISION ADD |
| MR | 07 | DSB | DOUBLE PRECISION SUBTRACT |

**INPUT/OUTPUT**

| IO | 14 | OCP | OUTPUT CONTROL PULSE |
| IO | 34 | SKS | SKIP IF SET |
| IO | 54 | INA | INPUT TO A |
| IO | 74 | OTA | OUTPUT FROM A |
| G  | 000511 | ISI | INPUT SERIAL INTERFACE TO A |
| G  | 000515 | OSI | OUTPUT SERIAL INTERFACE FROM A |
| IO | 74 | SMK | SET INTERRUPT MASK |

**CONTROL**

| G  | 000000 | HLT | HALT |
| G  | 000001 | NOP | NO OPERATION |
| G  | 140600 | SCL | SET C-BIT |
| G  | 140200 | RCB | RESET C-BIT |
| G  | 000005 | SGL | ENTER SINGLE PRECISION MODE |
| G  | 000007 | DRL | ENTER DOUBLE PRECISION MODE |
| G  | 000503 | EMC | ENTER MACHINE CHECK MODE |
| G  | 000501 | LMC | LEAVE MACHINE CHECK MODE |
| G  | 000021 | RMC (RMP) | RESET MACHINE CHECK |
| G  | 000011 | E16S (DXA) | ENTER 16K SECTOR ADDRESSING MODE |
| G  | 000013 | E32S (EXA) | ENTER 32K SECTOR ADDRESSING MODE |
| G  | 001013 | E32R | ENTER 32K RELATIVE ADDRESSING MODE |
| G  | 000505 | SVC | SUPERVISOR CALL |
| G  | 000311** | VIRY | VERIFY |
Table 3-1 (Cont)

**LOGICAL**

<table>
<thead>
<tr>
<th>MR</th>
<th>03</th>
<th>ANA</th>
<th>AND TO A</th>
</tr>
</thead>
<tbody>
<tr>
<td>MR</td>
<td>05</td>
<td>ERA</td>
<td>EXCLUSIVE OR TO A</td>
</tr>
<tr>
<td>G</td>
<td>140401</td>
<td>CMA</td>
<td>COMPLEMENT A</td>
</tr>
<tr>
<td>G</td>
<td>140413</td>
<td>LEO</td>
<td>CONVERT A=0 TO TRUE</td>
</tr>
<tr>
<td>G</td>
<td>140412</td>
<td>LNE</td>
<td>CONVERT (A=0) TO TRUE</td>
</tr>
<tr>
<td>G</td>
<td>140411</td>
<td>LLE</td>
<td>CONVERT A&lt;0 TO TRUE</td>
</tr>
<tr>
<td>G</td>
<td>140414</td>
<td>LGE</td>
<td>CONVERT A&gt;0 TO TRUE</td>
</tr>
<tr>
<td>G</td>
<td>140410</td>
<td>LLT</td>
<td>CONVERT A&lt;0 TO TRUE</td>
</tr>
<tr>
<td>G</td>
<td>140415</td>
<td>LGT</td>
<td>CONVERT A&gt;0 TO TRUE</td>
</tr>
</tbody>
</table>

**INTERRUPT**

| G   | 000401 | ENB       | ENABLE INTERRUPT |
| G   | 001001 | INH       | INHIBIT INTERRUPT |
| G   | 000415 | FSIM      | ENTER STANDARD INTERRUPT MODE |
| G   | 000417 | FVIM      | ENTER VECTORED INTERRUPT MODE |
| G   | 000411 | CAI       | CLEAR ACTIVE INTERRUPT |
| G   | 000043 | INK       | TRANSFER (INPUT) STATUS KEYS TO A |
| G   | 000405 | OTK       | TRANSFER (OUTPUT) A TO STATUS KEYS |

**SHIFT**

<p>| SH  | 0414NN | ALL (LGL) | A LEFT LOGICAL |
| SH  | 0404NN | ARL (LGR) | A RIGHT LOGICAL |
| SH  | 0416NN | ARL      | A LEFT ROTATE |
| SH  | 0406NN | ARR      | A RIGHT ROTATE |
| SH  | 0415NN | ALS      | A LEFT SHIFT |
| SH  | 0405NN | ARS      | A RIGHT SHIFT |
| SH  | 0410NN | LLL      | LONG LEFT LOGICAL |
| SH  | 0400NN | LRL      | LONG RIGHT LOGICAL |
| SH  | 0412NN | LLR      | LONG LEFT ROTATE |
| SH  | 0402NN | LRR      | LONG RIGHT ROTATE |
| SH  | 0411NN | LL.S     | LONG LEFT SHIFT |</p>
<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0401NN</td>
<td>LONG RIGHT SHIFT</td>
</tr>
<tr>
<td>000101</td>
<td>NORMALIZE</td>
</tr>
<tr>
<td>000041</td>
<td>TRANSFER SHIFT COUNTER TO A</td>
</tr>
</tbody>
</table>

**BYTE MANIPULATION**

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>141340</td>
<td>INTERCHANGE BYTES OF A</td>
</tr>
<tr>
<td>141140</td>
<td>INTERCHANGE BYTES OF A AND CLEAR LEFT BYTE</td>
</tr>
<tr>
<td>141240</td>
<td>INTERCHANGE BYTES OF A AND CLEAR RIGHT BYTE</td>
</tr>
<tr>
<td>141050</td>
<td>CLEAR LEFT BYTE OF A</td>
</tr>
<tr>
<td>141044</td>
<td>CLEAR RIGHT BYTE OF A</td>
</tr>
</tbody>
</table>

**TRANSFER AND SKIP**

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>01</td>
<td>UNCONDITIONAL JUMP</td>
</tr>
<tr>
<td>10</td>
<td>JUMP TO EA + 1 AND STORE P IN EA</td>
</tr>
<tr>
<td>100000</td>
<td>UNCONDITIONAL SKIP</td>
</tr>
<tr>
<td>12</td>
<td>INCREMENT, REPLACE MEMORY AND SKIP</td>
</tr>
<tr>
<td>140114</td>
<td>INCREMENT, REPLACE INDEX AND SKIP</td>
</tr>
<tr>
<td>140210</td>
<td>DECREMENT REPLACE INDEX AND SKIP</td>
</tr>
<tr>
<td>11</td>
<td>COMPARE A WITH MEMORY</td>
</tr>
<tr>
<td>140214</td>
<td>COMPARE A WITH ZERO</td>
</tr>
<tr>
<td>100400</td>
<td>SKIP ON A PLUS</td>
</tr>
<tr>
<td>101400</td>
<td>SKIP ON A MINUS</td>
</tr>
<tr>
<td>100040</td>
<td>SKIP ON A ZERO</td>
</tr>
<tr>
<td>101040</td>
<td>SKIP ON A NOT ZERO</td>
</tr>
<tr>
<td>100220</td>
<td>SKIP ON A GREATER THAN ZERO</td>
</tr>
<tr>
<td>101220</td>
<td>SKIP ON A LESS THAN OR EQUAL TO ZERO</td>
</tr>
<tr>
<td>100100</td>
<td>SKIP ON A BIT 16 ZERO</td>
</tr>
<tr>
<td>101100</td>
<td>SKIP ON A BIT 16 ONE</td>
</tr>
<tr>
<td>101260+N</td>
<td>SKIP ON A BIT N SET</td>
</tr>
<tr>
<td>100260+N</td>
<td>SKIP ON A BIT N RESET</td>
</tr>
<tr>
<td>100101</td>
<td>SKIP ON C-BIT SET</td>
</tr>
<tr>
<td>100001</td>
<td>SKIP ON C-BIT RESET</td>
</tr>
<tr>
<td>101200</td>
<td>SKIP ON MACHINE CHECK SET</td>
</tr>
<tr>
<td>100200</td>
<td>SKIP ON MACHINE CHECK RESET</td>
</tr>
</tbody>
</table>
Table 3-1 (Cont)

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>G</td>
<td>101036 SSS</td>
</tr>
<tr>
<td>G</td>
<td>100036 SSR</td>
</tr>
<tr>
<td>BR</td>
<td>101240+N SSN</td>
</tr>
<tr>
<td>BR</td>
<td>100240+N SRN</td>
</tr>
</tbody>
</table>

NOTES
-----

* DOUBLE PRECISION MODE MUST BE IN EFFECT (SEE DBL, SGL)

** EXECUTED BY TRAP TO SUBROUTINE

( ) = ALTERNATE MNEMONIC FOR COMPATIBILITY WITH OTHER ASSEMBLERS

CLASS CODES:

BR - BIT REFERENCE
G - GENERIC
IO - INPUT / OUTPUT
MR - MEMORY REFERENCE
SH - SHIFT
Variable Field

All except the generic instructions require an entry in the variable field that can be evaluated as a 16-bit single-precision quantity. The types of expression that can be used are summarized in Figure 3-1. If the expression is followed by ",1" (memory reference instructions only) the index bit is set.

For memory reference instructions, the variable field, indirect address bit, and indexing bit, interact to form the instruction's effective address.

Input/Output instructions interpret the variable field as the device code and function code of an I/O device controller.

For shift instructions, the variable field specifies the number of bit positions the A and B registers are to be shifted.

For bit reference instructions, the variable field specifies the panel sense switch (1-16) to be tested.

Generic instructions ignore the variable field.

Asterisk (Current Location): An asterisk in the variable field represents the current value of the assembler location counter. The asterisk is used in address expressions that describe a displacement from the current location:

\[
\text{COUNT IRS ALPHA} \\
\text{JMP } *-1 \\
\ldots \\
\ldots \\
\text{JMP COUNT}
\]

Both JMP instructions point to the same location, but the one using the asterisk does so without using a symbolic name.

Double Asterisk (Initially Zero): A double asterisk in the variable field causes the assembler to load zeroes in the 9-bit address field and the sector bit. (Indexing and indirect addressing are normal.) This convention is used when the desired location is to be developed or modified by other instructions or is not known at the time of assembly. For example:
Figure 3-1. General Format of Instruction Statements

3-8
**Equals Sign ( Literals):** A literal is a constant preceded by an equals sign, as in:

```
02023: 02.02032 (0025) F3#9 LDA =100 (OCTAL)
```

The assembler associates the numerical value of each literal with the symbol used (‘100 in this case) and reserves a storage location for a constant of that value. Any later reference to a literal of the same value addresses the same reserved location, even if a different constant format is used:

```
02024: 02.02032 (0026) LDA =40 (HEXADECIMAL)
02025: 02.02032 (0027) LDA =64 (DECIMAL)
```

**Literals are self-defining.** The name of the literal identifies the values of the constant to anyone reading the listing, whereas names assigned to constant locations by **SET** or similar pseudo-operations are meaningful only to the original programmer:

```
K100 SET '100
LDA K100+3
```

Actual locations containing literals are not assigned until the assembler reaches a **FIN** or **END** pseudo-operation. All literals assigned up to that point are then assigned sequential locations. On the final assembly pass, the address fields of statements that reference the literals are filled with the appropriate locations:

```
(0019) *-----TEST OF LITERALS AND FIN PSEUDO-OP
02000: 02.02003 (0020) LDA =100
02001: 05.02004 (0021) ADD =‘100
02002: 04.02005 (0022) STA =‘X’100'
02003: 000144 (0023) FIN
02004: 000100
02005: 000400
02006: 02.02003 (0024) LDA =100
02007: 06.02004 (0025) ADD =‘100
02008: 04.02005 (0026) STA =‘X’100'
02009: 00027 (0027) FIN
02010: 02.02003 (0028) LDA =100
02011: 06.02004 (0029) ADD =101
02012: 04.02005 (0030) STA =‘X’102
02013: 00054 (0070) END
02054: 000145
02055: 000482
```

---

3-9
ASCII Literals: Literals can be set to equal the binary codes of one or two ASCII characters. The form = C'X' loads a character (X, for example) into the left-hand byte (bits 1-8), and loads a space character into bits 9-16:

The form =AXY is loaded as two characters (X and Y, for example), with X in the left-hand byte (bits 1-8) and Y in the right-hand byte (bits 9-16):

(For ASCII character codes see Appendix E.)
MEMORY REFERENCE INSTRUCTIONS

Memory reference instructions are assembled as shown in Figure 3-2 and the following listing examples.

```plaintext
001215: 01 00100 (0156) ALPA JMP 100
001216: 02 01215 (0157) LDA ALPA
001217: 03 01215 (0158) ANA* ALPA
001220: 24 01215 (0159) BETA STA ALPA, 1
001221: 65 01215 (0160) ERA* ALPA, 1
001222: 06 01215 (0161) ADD ALPA
001223: 07 01215 (0162) SUB BETA
001224: 10 01224 (0163) JST *
001225: 12 01220 (0164) IRS BETA
001226: 00 01220 (0165) P2E BETA
001227: 00 01220 (0166) *** BETA
001230: 11 01230 (0167) GAMA CAS GAMA
001231: 13 01230 (0168) IMA GAMA
001232: 16 00050 (0169) NPY 40
001233: 17 01220 (0170) DIV BETA
001234: 15 01220 (0171) STX BETA
001235: 35 01220 (0172) LDX BETA
001236: 02 01220 (0173) DLD BETA
001237: 04 01220 (0174) DST BETA
001240: 06 01220 (0175) DAD BETA
001241: 67 00000 (0176) DSB* 0, 1
```
Figure 3-2. Assembly and Loading of Memory Reference Instruction
Operation Field

Mnemonic: The operation field must include one of the PRIME 200 memory reference instruction mnemonics shown above and listed in Table 3-1.

Asterisk (Indirect Addressing): An asterisk following the mnemonic specifies that the instruction word's indirect address bit is to be set.

Triple Asterisk (Dummy Instruction): A triple asterisk in place of an instruction mnemonic is a pseudo-operation code that causes the assembler to form a memory reference instruction with an op-code of zero. Another asterisk may be added to specify indirect addressing. The variable field of such a statement is treated like any other memory reference instruction:

\[(0185) *\]
\[(0186) *-----VACANT OPERATORS\]
\[(01246: 00 0001? (0187) **** 10\]
\[(01247: 00 0001? (0188) *** 10\]
\[(01250: 00 0001? (0189) *** 10\]
\[(01251: 00 0001? (0190) * 10\]
\[(0191) *\]

Variable Field

The variable field of a memory reference instruction contains an address expression (symbolic address) and an optional indexing symbol (,1).

Symbolic Addresses: Addresses are specified by any constant, variable, literal, or expression that can be evaluated as a single-precision 16-bit number. The sign (bit 1) is disregarded, and the magnitude bits (2-16) are interpreted as a memory location in the range from 0 to 32,767.

Addresses may be processed further by the loader if relocatable load mode is specified by the REL pseudo-operation. After loading, the way the CPU interprets the address depends on the addressing mode, controlled by E16S, E32S, and E32R instructions.

3-13
Indexing (,1): Indexing is specified by a ",1" following the address expression (optional). The form ",0" is interpreted as non-indexing. Therefore the 1 or 0 can be replaced by an expression using relational operators that returns a value of 0 or 1. For example in the statement

```
 02030: 22 02014 (0050) EXAMPLE LDA ALPHA, TEST EQ. 5
 000005 (0021) TEST SET 5
```

indexing results because the variable TEST equals 5 at the time of assembly. This feature would be useful mainly for conditional assembly operations.

For the LDX command, the assembler set the index bit, and for STX, the assembler clears the index bit. Indexing cannot be specified in these instructions.

INPUT/OUTPUT INSTRUCTIONS

Input/Output instructions are assembled in the form shown in Figure 3-3. Label and comment processing is normal.

Input/Output instructions are identified by a 6-bit operation code that occupies the indirect bit and indexing bit positions. Therefore, indexing and indirect addressing are not permitted.

The variable field must contain a four-bit function code concatenated with six-bit device address code. The resulting 10-bit code is usually specified in octal notation, as in Appendix D, but any kind of constant, variable, or expression is acceptable if it can be converted into a meaningful 10-bit code.

Examples:

```
01204: 030100 (0141) OCP 100
01205: 070101 (0142) SKS 101
01206: 170102 (0143) SMK 102
01207: 130106 (0144) INR 105
01210: 170123 (0145) OTA 123
0146: *---I/O INSTRUCTIONS
```
Figure 3-3. Assembly and Loading of Input/Output Instruction
SHIFT INSTRUCTIONS

Shift instructions are assembled in the form shown in Figure 3-4. Label and comment processing is normal.

Shift instructions are identified by a 10-bit operation code that occupies the indirect bit and indexing bit positions. Therefore, indexing and indirect addressing are not permitted.

The variable field must contain an expression that can be evaluated as a positive number representing the number of shifts to be executed. (The assembler forms the 2's complement of the quantity before setting it into bit position 11-16 of the instruction word supplied to the loader.) Any variables in the expression must be defined numerically by EQU or SET pseudo-operations (Section 4).

Examples:

\[ 01166: \quad 041400 \ (0125) \] \[ 01167: \quad 041400 \ (0126) \] \[ 01170: \quad 040577 \ (0127) \] \[ 01171: \quad 048676 \ (0128) \] \[ 01172: \quad 048475 \ (0129) \] \[ 01173: \quad 048475 \ (0130) \] \[ 01174: \quad 041574 \ (0131) \] \[ 01175: \quad 041673 \ (0132) \] \[ 01176: \quad 040072 \ (0133) \] \[ 01177: \quad 040071 \ (0134) \] \[ 01200: \quad 040276 \ (0135) \] \[ 01201: \quad 041067 \ (0136) \] \[ 01202: \quad 041166 \ (0137) \] \[ 01203: \quad 041265 \ (0138) \]
Figure 3-4. Assembly and Loading of Shift Instructions
BIT REFERENCE INSTRUCTIONS

Bit reference instructions test the condition of the panel sense switches; they are assembled as shown in Figure 3-5. Label and comment processing is normal.

Bit reference instructions are identified by a 12-bit operation code that occupies the indirect and indexing bit positions. Therefore these operations are not permitted.

The variable field must contain an expression that can be evaluated as a positive number between 1 and 16 decimal. The number becomes the code that selects the sense switch to be tested. Any variables in the expression must be defined numerically by EQU or SET pseudo-operations (Section 4).

Examples:

```
01211: 101246 (0146)  SSN 1
01212: 100244 (0144)  SR  2
01213: 101276 (0150)  SAS 15
01214: 100277 (0151)  SAR 16
```
Figure 3-5. Assembly and Loading of Bit Reference Instructions
GENERAL INSTRUCTIONS

Generic instructions (Figure 3-6) are fully defined by their operation codes and do not require operands, addresses, or other arguments in the variable field. The variable field must be null. Labels and comments are handled normally.

Examples:

```
01015: 000000 (0016) STRT HLT
01014: 000000 (0017) NOP
01013: 000005 (0018) SQL
01012: 000007 (0019) D6L
01011: 000011 (0020) DXA
01010: 000011 (0021) E165
01009: 000013 (0022) EXA
01008: 000013 (0023) E325
01007: 000021 (0026) RNP
01006: 000024 (0027) RNC
01005: 000041 (0028) SCA
01004: 000043 (0029) INK
01003: 000101 (0030) NRM
01002: 000105 (0031) EFXM
01001: 000107 (0032) LFXM
01000: 000107 (0033) CEA
00125: 000115 (0034) LJIM
00124: 000117 (0035) EJIM
00123: 000201 (0036) IAB
00122: 000205 (0037) PIM
00121: 000211 (0038) PID
00120: 000215 (0039) LPMJ
00119: 000217 (0040) EBNJ
00118: 000217 (0041) EVNJ
00117: 000311 (0042) DIAG
00116: 000401 (0044) ENB
00115: 000405 (0045) O6K
00114: 000411 (0046) CD4
00113: 000415 (0047) ESIM
00112: 000417 (0048) EVIN
00111: 000501 (0049) LMCM
00110: 000503 (0050) ENCM
00109: 000505 (0051) SVC
00108: 000511 (0052) ISI
00107: 000515 (0053) OSI
00106: 001001 (0054) INH
00105: 001011 (0055) E64R
00104: 001013 (0056) E32R
```
Figure 3-6. Assembly and Loading of Generic Instructions
| 01066 | 081400 (0059) | ERH |
| 01067 | 100000 (0060) | SKP |
| 01069 | 101400 (0061) | SLT |
| 01071 | 101400 (0062) | SHT |
| 01072 | 100400 (0063) | SGE |
| 01073 | 100400 (0064) | SPL |
| 01074 | 101200 (0065) | SLE |
| 01075 | 100200 (0066) | SGT |
| 01076 | 100400 (0067) | SEO |
| 01077 | 100000 (0068) | SIE |
| 01100 | 101000 (0069) | SNE |
| 01101 | 101000 (0070) | SNZ |
| 01102 | 101000 (0071) | SNC |
| 01103 | 100000 (0072) | SRC |
| 01104 | 101200 (0073) | SSNC |
| 01105 | 101200 (0074) | SPS |
| 01106 | 100200 (0075) | SRNC |
| 01107 | 100200 (0076) | SPN |
| 01110 | 100100 (0077) | SLZ |
| 01111 | 101100 (0078) | SLN |
| 01112 | 101030 (0079) | SS1 |
| 01113 | 100000 (0080) | SR1 |
| 01114 | 101030 (0081) | SS2 |
| 01115 | 100010 (0082) | SR2 |
| 01116 | 101000 (0083) | SS3 |
| 01117 | 100000 (0084) | SR3 |
| 01120 | 101000 (0085) | SS4 |
| 01121 | 100000 (0086) | SR4 |
| 01122 | 101030 (0087) | SS5 |
| 01123 | 100000 (0088) | SSR |
| 01124 | 140000 (0089) | CRA |
| 01125 | 140024 (0090) | CHS |
| 01126 | 140100 (0091) | SSP |
| 01127 | 140200 (0092) | RCB |
| 01130 | 140320 (0093) | CSB |
| 01131 | 140400 (0094) | CNA |
| 01132 | 140400 (0095) | TCA |
| 01133 | 140500 (0096) | SSM |
| 01134 | 140500 (0097) | SCB |
| 01135 | 141044 (0098) | CAR |
| 01136 | 141050 (0099) | CSL |
| 01137 | 141146 (0100) | ICL |
| 01140 | 141200 (0101) | ICR |
| 01141 | 141200 (0102) | ASA |
| 01142 | 141200 (0103) | A1A |
| 01143 | 141210 (0104) | ACA |
| 01144 | 141300 (0105) | ICA |
| 01145 | 140010 (0106) | CRL |
| 01146 | 140014 (0107) | CRB |
| 01147 | 140104 (0108) | XCA |
| 01150 | 140100 (0109) | SOA |
| 01151 | 140100 (0110) | SIA |
| 01152 | 140114 (0111) | IRX |
| 01153 | 140204 (0112) | XCB |
| 01154 | 140210 (0113) | DRX |
| 01155 | 140214 (0114) | CAZ |

3-22
<table>
<thead>
<tr>
<th>01156</th>
<th>140304 (0115)</th>
<th>A2A</th>
</tr>
</thead>
<tbody>
<tr>
<td>01157</td>
<td>140310 (0116)</td>
<td>S2A</td>
</tr>
<tr>
<td>01160</td>
<td>140410 (0117)</td>
<td>LLT</td>
</tr>
<tr>
<td>01161</td>
<td>140411 (0118)</td>
<td>LLE</td>
</tr>
<tr>
<td>01162</td>
<td>140412 (0119)</td>
<td>LNE</td>
</tr>
<tr>
<td>01163</td>
<td>140413 (0120)</td>
<td>LEO</td>
</tr>
<tr>
<td>01164</td>
<td>140414 (0121)</td>
<td>LGE</td>
</tr>
<tr>
<td>01165</td>
<td>140415 (0122)</td>
<td>LGT</td>
</tr>
</tbody>
</table>

3-23
SECTION 4

PSEUDO-OPERATIONS

Pseudo-operation statements are commands (or directives) to the assembler or loader, rather than instructions to be assembled and executed in a user's program. Various classes of pseudo-operation are provided, to control the assembly and load modes, assign values to symbols and data constants, define macros, link programs, allocate storage, and control conditional assembly. The mnemonics of all the PRIME 200 assembler pseudo-operations are listed in Table 4-1. Pseudo operations are described in this section according to class, except for those used in Macro definitions (Section 5).

STATEMENT FORMAT

Pseudo-operations have an operation field and a variable field separated by spaces, the backslash tab character, or a comma ((see Figure 4-1). In addition, some pseudo-operations require a label to be present or absent. Therefore the statement format description in the following paragraphs includes the label field.

Constants, variables, and expressions used in pseudo-operations conform to the general features defined in Section 2.

The operation field contains the mnemonic that identifies the pseudo-operation.

The variable field may contain one or more arguments, separated by single spaces or commas. Arguments may be constants, variables, or expressions as defined in Section 2. Arguments for certain operations such as BCI may also consist of ASCII character strings. (Spaces and commas occurring within such strings are not interpreted as argument delimiters.)

Symbolic names or other variables used in the variable field must be previously defined, unless otherwise stated in the pseudo-operation definition.

Address expressions are evaluated as single-precision values and used as an absolute 16-bit memory address. If the relocatable mode is in effect during loading, the relocation factor is added to the address. Certain statements (DAC, XAC, *** ) accept the indirect address (*) and indexing (,1) symbols. These are interpreted according to whether the extended addressing mode is in effect. (See EXD and LXD pseudo-operations.)

4-1
<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Definition</th>
<th>Class*</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABS</td>
<td>Set Mode to Absolute</td>
<td>AS</td>
</tr>
<tr>
<td>BACK (TO)</td>
<td>Loop Back (Macros Only)</td>
<td>MA</td>
</tr>
<tr>
<td>BCI</td>
<td>Define ASCII String</td>
<td>DA</td>
</tr>
<tr>
<td>BES</td>
<td>Define Block Ending with Symbol</td>
<td>ST</td>
</tr>
<tr>
<td>BSS</td>
<td>Define Block Starting with Symbol</td>
<td>ST</td>
</tr>
<tr>
<td>BSZ</td>
<td>Define Block Set to Zeros</td>
<td>ST</td>
</tr>
<tr>
<td>CALL</td>
<td>External Subroutine Reference</td>
<td>ST</td>
</tr>
<tr>
<td>CF1-CF5</td>
<td>Ignored (Provided for Compatibility with Other Assemblers)</td>
<td>AS</td>
</tr>
<tr>
<td>COMN</td>
<td>Define Common Items</td>
<td>ST</td>
</tr>
<tr>
<td>DAC</td>
<td>Local Address Definition</td>
<td>DA</td>
</tr>
<tr>
<td>DATA</td>
<td>Set Data Constant</td>
<td>DA</td>
</tr>
<tr>
<td>DBP</td>
<td>Set Double Precision Constant</td>
<td>DA</td>
</tr>
<tr>
<td>DEC</td>
<td>Set Decimal Constant</td>
<td>DA</td>
</tr>
<tr>
<td>ENT</td>
<td>Define External Entry Points</td>
<td>ST</td>
</tr>
<tr>
<td>GO (TO)</td>
<td>Forward Reference</td>
<td>AS</td>
</tr>
<tr>
<td>EJCT</td>
<td>Eject Page (Start New Page)</td>
<td>LI</td>
</tr>
<tr>
<td>ELSE</td>
<td>Reverse Conditional Assembly</td>
<td>CO</td>
</tr>
<tr>
<td>END</td>
<td>End of Source Statements</td>
<td>A</td>
</tr>
<tr>
<td>ENDC</td>
<td>End Conditional Assembly Area</td>
<td>CO</td>
</tr>
<tr>
<td>ENDM</td>
<td>End of Macro Definition</td>
<td>MA</td>
</tr>
<tr>
<td>EQU</td>
<td>Define Variable</td>
<td>SY</td>
</tr>
<tr>
<td>EXD</td>
<td>Enter Extended Addressing Mode</td>
<td>LO</td>
</tr>
<tr>
<td>EXT</td>
<td>Flag External References</td>
<td>ST</td>
</tr>
<tr>
<td>FAIL</td>
<td>Force Error Message</td>
<td>SP</td>
</tr>
<tr>
<td>FIN</td>
<td>Insert Literals</td>
<td>AS</td>
</tr>
<tr>
<td>HEX</td>
<td>Set Hexadecimal Constants</td>
<td>DA</td>
</tr>
<tr>
<td>IF</td>
<td>Conditional Statement</td>
<td>CO</td>
</tr>
<tr>
<td>IFM</td>
<td>Continue Assembly if Minus</td>
<td>CO</td>
</tr>
<tr>
<td>IFN</td>
<td>Continue Assembly if Non-Zero</td>
<td>CO</td>
</tr>
<tr>
<td>IFP</td>
<td>Continue Assembly if Plus</td>
<td>CO</td>
</tr>
<tr>
<td>IFZ</td>
<td>Continue Assembly if Zero</td>
<td>CO</td>
</tr>
<tr>
<td>List</td>
<td>Enable Listing</td>
<td>LI</td>
</tr>
<tr>
<td>LSMD</td>
<td>List Macro Expansions (Data Statements Only)</td>
<td>MA</td>
</tr>
<tr>
<td>LSTM</td>
<td>List Macro Expansions (All Statements)</td>
<td>MA</td>
</tr>
<tr>
<td>LXD</td>
<td>Leave Extended Addressing Mode</td>
<td>LO</td>
</tr>
<tr>
<td>MAC</td>
<td>Start Macro Definition</td>
<td>MA</td>
</tr>
<tr>
<td>MOR</td>
<td>More Input Required</td>
<td>AS</td>
</tr>
<tr>
<td>NLSM</td>
<td>No Listing of Macro Expansions</td>
<td>MA</td>
</tr>
<tr>
<td>NLST</td>
<td>Inhibit Listing</td>
<td>LI</td>
</tr>
<tr>
<td>OCT</td>
<td>Define Octal Constants</td>
<td>DA</td>
</tr>
<tr>
<td>ORG</td>
<td>Define Origin Location</td>
<td>AS</td>
</tr>
<tr>
<td>SAY</td>
<td>List Message to Operator (Within Macro Definitions)</td>
<td>MA</td>
</tr>
<tr>
<td>SET</td>
<td>Redefine a Variable</td>
<td>SY</td>
</tr>
<tr>
<td>SETB</td>
<td>Set Base Sector</td>
<td>LO</td>
</tr>
<tr>
<td>SETC</td>
<td>Set Common Base Address</td>
<td>ST</td>
</tr>
<tr>
<td>SUBR</td>
<td>Define Entry Points</td>
<td>ST</td>
</tr>
<tr>
<td>REL</td>
<td>Set Mode to Relocatable</td>
<td>AS</td>
</tr>
<tr>
<td>VFD</td>
<td>Define Variable Fields</td>
<td>DA</td>
</tr>
<tr>
<td>XAC</td>
<td>Define External Address</td>
<td>DA</td>
</tr>
<tr>
<td>***</td>
<td>Dummy Memory Reference Instructions</td>
<td>DA</td>
</tr>
</tbody>
</table>

* CLASSES: AS - Assembly Control  LO - Loader Control
CO - Conditional Assembly  MA - Macro Definition
DA - Data Defining  ST - Storage Allocation
LI - Listing Control  SY - Symbol Defining
Figure 4-1. General Format of Pseudo-Operation Statements
ASSEMBLY CONTROLLING PSEUDO OPERATIONS

ABS (Set Mode to Absolute)

<table>
<thead>
<tr>
<th>Label</th>
<th>Operation Field</th>
<th>Variable Field</th>
</tr>
</thead>
<tbody>
<tr>
<td>Optional</td>
<td>ABS</td>
<td>Must be vacant</td>
</tr>
</tbody>
</table>

Sets to absolute the assembly and loading mode of all subsequent memory reference instructions. ABS may be terminated by REL and vice-versa. The ABS mode is the normal default mode of assembly.

REL (Set Mode to Relocatable)

<table>
<thead>
<tr>
<th>Label</th>
<th>Operation Field</th>
<th>Variable Field</th>
</tr>
</thead>
<tbody>
<tr>
<td>Not Used</td>
<td>REL</td>
<td>Must be vacant</td>
</tr>
</tbody>
</table>

Sets to relocatable the assembly and loading mode of all subsequent memory reference instructions. REL may be terminated by ABS.

ORG (Define Origin Location)

<table>
<thead>
<tr>
<th>Label</th>
<th>Operation Field</th>
<th>Variable Field</th>
</tr>
</thead>
<tbody>
<tr>
<td>Optional</td>
<td>ORG</td>
<td>Address Expression</td>
</tr>
</tbody>
</table>

Sets up a new assembler location count equal to the value of the address expression. This new origin is considered absolute or relocatable depending on the current mode of the assembler and loader. In absolute mode, program loading continues at the location specified by the address expression. In relocatable mode, program loading continues at the location specified by the address expression plus the loader's relocation factor.

If the statement includes a label, the label variable is set equal to the location count before the ORG is executed.
Examples:

(0001) * DEMONSTRATES REL AND ABS

00000: 000001 (0003)
00001: 000001 (0004)
00002: 00 00607 (0005)
00003: 15 00610 (0006)
00004: 12 00611 (0007)
00005: 01 00600 (0008)
       00600 (0009)
00006: 02 00612 (0011) ABSO
00007: 00 00603 (0012)
00008: 01 00604 (0013)
00009: 000000 (0015)
00010: 02 00607 (0016) Y
00011: 02 00614 (0017)
00012: 000000 (0018)
00013: 000000 (0019) SAVX
00014: 000000 (0020) SAVX
00015: 000000 (0021) CTR
       00612 (0022)

REL
NOP
STA SAVA
STX SAVX
IRS CTR
JMP ABSO
ORG '600
ABS
LDA =1
STA **2
JMP Y
REL
LDA SAVA
LDA SAVX
HLT
DATA 0
DATA 0
DATA 0
END

START RELOCATABLE
SAVE REGISTERS
UPDATE COUNTER
JUMP TO ABSOLUTE LOCATION
STARTS AT LOCATION '600
RETURN TO RELOCATABLE
RESTORE REGISTERS
FIN (Insert Literals)

<table>
<thead>
<tr>
<th>Label</th>
<th>Operation Field</th>
<th>Variable Field</th>
</tr>
</thead>
<tbody>
<tr>
<td>Optional</td>
<td>FIN</td>
<td>Not used</td>
</tr>
</tbody>
</table>

All literals defined since the beginning of the program (or the last FIN statement) are assembled into a "literal pool", starting at the current location count. Processing of subsequent statements begins at the first location count following the literals. FIN performs the same functions as the END statement, but does not terminate the assembly. By using FIN, the programmer can distribute literals throughout the program, and possibly reduce the number of cross-sector indirect address links that must be formed by the loader. (However it is important to make sure that the program will jump over the pool of literals and not attempt to execute them as instructions.)

MOR (More Input Required)

<table>
<thead>
<tr>
<th>Label</th>
<th>Operation Field</th>
<th>Variable Field</th>
</tr>
</thead>
<tbody>
<tr>
<td>Optional</td>
<td>MOR</td>
<td>Not used</td>
</tr>
</tbody>
</table>

When entered as the last statement on a source tape, MOR causes the input device to stop. A continuation tape can then be mounted. When the computer START switch is pressed, assembly continues with the first statement on the continuation tape.

END (End of Source Statements)

<table>
<thead>
<tr>
<th>Label</th>
<th>Operation Field</th>
<th>Variable Field</th>
</tr>
</thead>
<tbody>
<tr>
<td>Optional</td>
<td>END</td>
<td>Address expression</td>
</tr>
</tbody>
</table>

Terminates processing of the source program. All literals accumulated since the beginning of the program (or the last FIN statement) are assigned locations starting at the current location count.
In a two-pass assembly, the computer halts on the first pass when the END statement is reached. The operator must then return the source tape to its starting point and restart the computer to begin pass two. (New assembly parameters can be specified on the second pass, if additional outputs are required.)

When the END statement is reached on the second pass, the address expression is included in the object text for action by the loader, which can be directed to start program execution at the specified location. If the address field is null, the starting location is assumed to be the first location of the program.

CF1 Through CF5

Pseudo-operations CF1 through CF5 have no effect on this assembler. However, these statements are accepted without generating error messages, in order to maintain compatibility with other assemblers.

GO, GO TO (Forward Reference)

<table>
<thead>
<tr>
<th>Label</th>
<th>Operation Field</th>
<th>Variable Field</th>
</tr>
</thead>
<tbody>
<tr>
<td>Not used</td>
<td>GO or GO TO</td>
<td>Statement label</td>
</tr>
</tbody>
</table>

Assembly is suspended for all statements following this one until a statement having the specified label is found. The GO (GO TO) statement must point forward to a statement label that is not yet defined. An error condition exists if the assembler reaches an END, MAC, or ENDM statement before finding the specified label.

Examples

GO TO K31

GO T174

IF (OPTION .EQ. 3) GO TO AL28

LDA X : ADD Y : GO TO Z20
LISTING CONTROL PSEUDO-OPERATIONS

LIST (Enable Listing)

<table>
<thead>
<tr>
<th>Label</th>
<th>Operation Field</th>
<th>Variable Field</th>
</tr>
</thead>
<tbody>
<tr>
<td>Not used</td>
<td>LIST</td>
<td>Not used</td>
</tr>
</tbody>
</table>

Causes all statements to be listed except those generated by macro expansion. This is the assembler's default mode - a LIST statement is not needed unless a NLST statement has previously inhibited listing.

NLST (Inhibited Listing)

<table>
<thead>
<tr>
<th>Label</th>
<th>Operation Field</th>
<th>Variable Field</th>
</tr>
</thead>
<tbody>
<tr>
<td>Not used</td>
<td>NLST</td>
<td>Not used</td>
</tr>
</tbody>
</table>

Inhibits listing of all subsequent statements until a LIST statement is encountered. LIST and NLST may be used together in source text for selective control over the sections to be listed. The LSTM, LSMD, and NLSM statements provide control of listing for macro definitions; for details, see Section 5.

EJCT (Eject Page)

<table>
<thead>
<tr>
<th>Label</th>
<th>Operation Field</th>
<th>Variable Field</th>
</tr>
</thead>
<tbody>
<tr>
<td>Not used</td>
<td>EJCT</td>
<td>Not used</td>
</tr>
</tbody>
</table>

Causes the listing device to eject the page (execute a form feed), print the current page title and page number, and feed two blank lines before resuming listing. This function is operable only with devices which have a mechanical form feed capability, such as a line printer.
LOADER CONTROLLING PSEUDO OPERATIONS

The following statements generate special messages in the object text that provide control information to the linking loader.

**EXD (Enter Extended Addressing Mode)**

<table>
<thead>
<tr>
<th>Label</th>
<th>Operation Field</th>
<th>Variable Field</th>
</tr>
</thead>
<tbody>
<tr>
<td>Optional</td>
<td>EXD</td>
<td>Not used</td>
</tr>
</tbody>
</table>

Notifies the loader that extended (32K) addressing mode is in effect. The loader processes subsequent indirect address words as having a 15-bit address field and an indirect bit, but no index bit. The CPU must be set to extended addressing mode by an E32S instruction.

**LXD (Leave Extended Addressing Mode)**

<table>
<thead>
<tr>
<th>Label</th>
<th>Operation Field</th>
<th>Variable Field</th>
</tr>
</thead>
<tbody>
<tr>
<td>Optional</td>
<td>LXD</td>
<td>Not used</td>
</tr>
</tbody>
</table>

Causes loader to leave extended mode and resume 16K addressing mode (the normal default mode of the loader). In this mode the loader processes indirect address words as having a 14 bit address field, an indirect bit, and an index bit. However, the operator can override the LXD mode during loading, and force extended addressing.

**SETB (Set Base Sector)**

<table>
<thead>
<tr>
<th>Label</th>
<th>Operation Field</th>
<th>Variable Field</th>
</tr>
</thead>
<tbody>
<tr>
<td>Optional</td>
<td>SETB</td>
<td>Address expression</td>
</tr>
</tbody>
</table>
Specifies a base sector and starting address for cross-sector indirect address links.

Normally the loader generates address links starting at location '100 of Sector zero. This statement permits the loader to generate some address links in the same sector as the program which refers to them. Memory locations to be used for this purpose must be reserved by the program.

Examples:

005000 (0003) ORG '5000 START LINKS AT BEGINNING OF SECTOR 5
05000: 01.05025 (0004) JMP ++21 JUMP OVER LINKS
005001 (0005) SETB * ALLOCATE 20 LOCATIONS FOR ADDRESS LINKS STARTING AT '5001
05025: (0006) ESS 20

The first SETB pseudo-op for a given base sector determines the location at which the indirect word table will begin in that sector. The table then grows upward in successively higher locations. Other SETB pseudo-ops referencing the same sector do not re-origin the table for that sector --- table filling resumes where it left off. During loading, the B-Register setting may be used to assign a starting address for the links; if so the B-Register setting is treated like a SETB pseudo-operation preceding the first word to be loaded.

At the end of each subprogram, the base sector reverts to sector zero. The loader retains knowledge of the last location used in each base sector. When the base sector reverts to zero, no indirect words are lost.

Note that in general cross-sector reference pools may grow unpredictably and overwrite program areas during loading, so that extreme care must be used in assigning SETB areas.
DATA DEFINING PSEUDO-OPERATIONS

This group of pseudo-operations is used to initialize memory locations to known starting values. Data and address constants may be specified in a variety of formats, for coding convenience. Simple coding conventions allow the programmer to use ASCII, hexadecimal, octal, or fixed and floating point decimal notation to specify constant values. The assembler interprets the notation and automatically generates one, two, or more data words in the proper internal binary format for single or double precision, fixed or floating point arithmetic.

DATA (Set Data Constant)

This is the basic PRIME 200 pseudo-operation for presetting memory locations to equal expressions, ASCII strings, or numerical constants. Constants can be expressed in decimal, octal, or hexadecimal form. Decimal quantities can be specified in single or double precision, fixed or floating points, formats. The basic format of the DATA statement is:

<table>
<thead>
<tr>
<th>Label</th>
<th>Operation Field</th>
<th>Variable Field</th>
</tr>
</thead>
<tbody>
<tr>
<td>Optional</td>
<td>DATA</td>
<td>One or more expressions, ASCII strings, or numerical data constants</td>
</tr>
</tbody>
</table>

The current location is set equal to the expression(s) in the variable field. The variable field may contain any number of subfields, separated by commas. Subfields are assembled in consecutive locations starting with the leftmost subfield. If an expression requires more than one location (e.g. floating point), consecutive locations are used.

ASCII Strings: ASCII character strings are specified by the letter C followed by the string enclosed in apostrophes. ASCII characters so specified are packed two per word during assembly. Single characters are left-justified with the remainder of the word filled with zeroes. The number of characters per statement is not limited.

The string portion of a data statement cannot be continued on the next line. Within the string itself, the (!) character permits the assembler to encode restricted characters such as ' (end of string), <(start of macro arg. ref.) or CR (end of statement). Examples:

| 05026: | 140640 (0008) | DATA CA' |
| 05027: | 140702 (0009) | DATA CA'ABCDEF' |
| 05030: | 141704       | DATA CA' |
| 05031: | 142706       | DATA CA'AB'12' |
| 05032: | 140702 (0010) | DATA CA' |
| 05033: | 123661       | 4-11     |
| 05034: | 131240       |          |
Numerical Constants: The form in which a constant is specified determines whether the assembler will process it as single or double precision, fixed or floating point. The general format for numerical constants is:

\[
\begin{align*}
\text{Number} & \quad \text{Exponent} & \quad \text{Scaling} \\
(+) \text{ Number} & \quad E \quad (+) \text{ mm} & \quad B \quad (+) \text{ nn}
\end{align*}
\]

If the number part of the statement is a decimal integer or fraction, it can in some cases be modified by a decimal exponent (E for single precision, EE for double precision) or a binary scaling factor (B for single precision, BB for double precision). Table 4-2 summarizes the legal combinations of number, exponent, and scaling designators.

Fixed Point Single Precision: Constants in fixed point single precision format are assembled to form a sign bit and 15 magnitude bits, as shown in Figure 4-2A. The CPU internally treats such arithmetic quantities as binary fractions ranging between -1 and slightly less than +1. The assembler, however, handles single precision words as signed integers ranging between -32,768 and +32,767. Constants in DATA statements may be expressed as integers within that range, using decimal, octal, or hexadecimal notation.

Expressions must be capable of being evaluated as single-precision constants only. Variables used in expressions must be previously defined.

Examples:

<table>
<thead>
<tr>
<th>Hexadecimal</th>
<th>Octal</th>
<th>Decimal</th>
<th>Expressions</th>
</tr>
</thead>
<tbody>
<tr>
<td>X'12AB'</td>
<td>0'1234'</td>
<td>12</td>
<td>X*2+3</td>
</tr>
<tr>
<td>X'-12AB'</td>
<td>0'-1234'</td>
<td>-12</td>
<td>ALPHA</td>
</tr>
<tr>
<td>$12AB</td>
<td>0'1234'</td>
<td>0</td>
<td>Y .AND. '77</td>
</tr>
<tr>
<td>$-12AB</td>
<td>'1234</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X'12AB</td>
<td>'-1234'</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$8000</td>
<td>'077777'</td>
<td>32767</td>
<td>1.23B6 (using binary scaling)</td>
</tr>
<tr>
<td></td>
<td>'100000'</td>
<td>-32768</td>
<td>1.23E3B12 (using decimal exponent and binary scaling)</td>
</tr>
<tr>
<td>Form of Number</td>
<td>Decimal Exponent (E or EE±mm)</td>
<td>Binary Scaling (B or BB±nn)</td>
<td>Assembler Interprets Constant As:</td>
</tr>
<tr>
<td>--------------------------------</td>
<td>-------------------------------</td>
<td>----------------------------</td>
<td>-----------------------------------</td>
</tr>
<tr>
<td>Expression using Symbolic Variables</td>
<td>--</td>
<td>--</td>
<td>Single Precision Fixed Point</td>
</tr>
<tr>
<td>Hexadecimal</td>
<td>--</td>
<td>--</td>
<td></td>
</tr>
<tr>
<td>Octal</td>
<td>--</td>
<td>--</td>
<td></td>
</tr>
<tr>
<td>Decimal Integer</td>
<td>--</td>
<td>--</td>
<td></td>
</tr>
<tr>
<td>Decimal Integer or Fraction</td>
<td>--</td>
<td>B</td>
<td>Double Precision Fixed Point</td>
</tr>
<tr>
<td></td>
<td>E</td>
<td>B</td>
<td></td>
</tr>
<tr>
<td></td>
<td>--</td>
<td>BB</td>
<td></td>
</tr>
<tr>
<td></td>
<td>EE</td>
<td>BB</td>
<td></td>
</tr>
<tr>
<td></td>
<td>EE</td>
<td>--</td>
<td>Double Precision Floating Point</td>
</tr>
<tr>
<td></td>
<td>E</td>
<td>--</td>
<td>Single Precision Floating Point</td>
</tr>
<tr>
<td>Decimal Fraction</td>
<td>--</td>
<td>--</td>
<td></td>
</tr>
</tbody>
</table>
A. DATA FORMAT

B. "B" CODES FOR BINARY SCALING

Figure 4-2. Single Precision Fixed Point Constants
Powers of 10 (E) and Binary Scaling (B): For single-precision decimals only, the E and B notation provides flexibility in scaling data constants. Expressions with binary scaling are formed by a decimal integer or fraction in the range from -32768 to +32767, followed by the letter B and an integer from -1 to +15.

Examples:

<table>
<thead>
<tr>
<th>Assembled As:</th>
<th>Decimal Equivalent</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 001 100.100 00 00 000 000 B6</td>
<td>6400</td>
</tr>
<tr>
<td>0 000 000 001 000 000 B8</td>
<td>64</td>
</tr>
<tr>
<td>0 000 001 010 000 000 B8</td>
<td>640</td>
</tr>
</tbody>
</table>

In general terms, a constant entered as $K_{10}Bn$ is converted to $K_2(2^{-n})$, where $K_{10}$ is the decimal constant, $K_2$ is the same constant expressed as a binary fraction, and $n$ is the number following the letter B. Positions for B values -1 through 15 are shown in Figure 4-2B. Any bits of the repositioned binary fraction that extend to the left or right of the 15 magnitude bits of the data word are truncated.

In the first example, the fraction 12.5 is converted to the binary value 1 100.1 and positioned in the 16 bit data word so that the binary point is at position B6. The result is equivalent to decimal 6400.

If an E code is present, the decimal value is multiplied by the power of 10 specified by the integer following the E before it is converted to binary. Thus a constant entered as $K_{10}E_mBn$ is converted as $K_2(10^m)(2^{-n})$. The exponent, $m$, may be negative (-) or positive (+ or unspecified).

In fixed-point single precision constant expressions, an exponent (E) cannot be used unless binary scaling (B) is also specified. If E is used alone (as in "5E2") the expression is interpreted as floating point (described later).
Fixed Point Double Precision: The assembler handles fixed point double precision words as integers ranging between \(-(2^{30})\) and \(+(2^{30} - 1)\). \((2^{30} = 1,073,741,824.)\) Such constants are assembled as two consecutive data words, in a format determined by the CPU's double precision arithmetic procedures. (See Figure 4-3A.) The first word must load in an even location; if the location count happens to be odd, one location is skipped. Negative numbers are represented in two's complement notation, but bit 1 of the second word is always 0.

When expressed in DATA statements, fixed-point double precision constants must include a binary scale factor (BBn). A decimal exponent (EEn) is optional.

The BB codes for binary scaling are interpreted in the same way as single precision B codes, but extend into the second word of precision as shown in Figure 4-3B. The EE code, if present, is interpreted in the same way as single precision E codes and can only be used when a BB code is also present.

**Examples:**

<table>
<thead>
<tr>
<th>Assembled as:</th>
<th>Decimal Equivalent</th>
</tr>
</thead>
<tbody>
<tr>
<td>12.5BB6 word 1 0 001 100 100 000 000</td>
<td>6400.00000</td>
</tr>
<tr>
<td>or 6.4EE3BB15 word 2 0 000 000 000 000 000</td>
<td></td>
</tr>
<tr>
<td>7BB16 word 1 0 000 000 000 000 011</td>
<td>3.50000</td>
</tr>
<tr>
<td>word 2 0 100 000 000 000 000</td>
<td></td>
</tr>
</tbody>
</table>

Bits of the scaled binary quantity that extend to the left of word 1 or right of word 2 are truncated.
A. DATA FORMAT

B. "BB" CODES FOR BINARY SCALING

Figure 4-3. Double Precision Fixed Point Constants
Single Precision Floating Point: Floating point data formats are defined by the procedures of the floating point math routines in the FORTRAN/Math Library. (See Figure 4-4.)

Single-precision floating point quantities are expressed by a decimal fraction, with or without decimal exponent (EEmm). (Binary scaling must not be specified.)

Examples:

```
05046: 042100 (0017)   DATA  1.28E2
05047: 000000
05050: 040321 (0018)
05051: 165605
05052: 137456 (0019)
05053: 012172
05054: 024563 (0020)
05055: 045342
```

The assembler converts the specified values to an 8-bit binary exponent and 23-bit binary fraction in two successive words, as shown in Figure 4-4A. The exponent is represented in excess-128 notation, and can range from $2^{-127}$ to $2^{127}$ (roughly $10^{-38}$ to $10^{+38}$). An error printout occurs if the exponent exceeds this range. The assembler automatically generates a normalized fraction of the largest possible value less than 1. Numbers specified in this format have about 6.8 significant decimal digits (+8,388,607).

Negative numbers are formed by generating a positive number of the specified magnitude and then forming the two's complement of both data words, including the exponent. The number zero is assembled as two consecutive all-zero data words.

Double Precision Floating Point: Double-precision floating point quantities are expressed by a decimal integer or fraction with a decimal exponent (EEmm). (Binary scaling must be specified.)

The assembler converts the specified value to an 8-bit binary exponent and 39-bit binary fraction, in three successive words, as shown in Figure 4-4B. The exponent is represented in the same excess-128 notation as single-precision floating point. The assembler automatically generates a normalized fraction of the largest possible value less than 1. Numbers specified in this format can have about 11.5 significant decimal digits (+549,755,000,000).

Negative numbers are formed by generating a positive number of the specified magnitude and then taking the two's complement of all three data words, including the exponent. The number zero is assembled as three consecutive all-zero data words.
Figure 4-4. Floating Point Word Formats
Examples:

```
05056:  042100 (0021)       DATA  1.28EE2
05057:  000000             DATA  1.28EE2
05060:  000000             DATA  1.28EE2
05061:  135700 (0022)      DATA  1.28EE2
05062:  000000             DATA  1.28EE2
05063:  000000             DATA  1.28EE2
05064:  024563 (0023)      DATA  1.28EE-14
05065:  045312             DATA  1.28EE-14
05066:  057537             DATA  1.28EE-14
```
Repeated Constants: Constants that do not start with a digit or a decimal point may be preceded with a repeat count "n" (positive integer) which will cause the value to be generated n times.

Examples:

3X'12AB'
9C'XX'
15'16
6(ALPHA+1)
3(1.5E6)
5(-0.012EE-3)

Multiple and Implied DATA Statements: A DATA statement can contain more than one constant, separated by commas. Constants are converted to the appropriate number of data words and loaded into consecutive memory cells starting with the current location count.

The assembler will process any statement that starts with a constant (not counting the optional label field) as an implied DATA statement.

Examples:

DATA 16
DATA 3, 10, -2, -3, 0, 0, 10, AP3
DATA 3, '12, X'-02', -X'3', 20'0', $A, AP3+2-1;
16, 3, 10, -2, XYZ-2
100
DATA -4, 1.23E4, +1176EE3BB24, 16(0.0)
DATA 4(X6*2-1)
Summary: The following examples show many varieties of DATA statements. Table 4-2 summarizes the legal combinations of constants, B and BB codes, and E and EE codes in numerical values.

05156: 000020 (0031) DATA 16
05157: 000003 (0032) DATA 3,10,-2,-3,0,0,10,AP3
05160: 000012
05161: 177775
05162: 177775
05163: 000000
05164: 000000
05165: 000012
05166: 005170
05167: 131640
05170 00 000000 (0033) AP3 DAC **
05171: 000003 (0034) DATA 3,'12,X'-02',-X'3',20'0',#A,AP3+2-1
05172: 000012
05173: 177775
05174: 177775
05175: 000000
05176: 000006
05177: 000000
05200: 000000
--- --- --- --- ---
05220: 000000
05221: 000012
05222: 005171
**DEC (Set Decimal Constant)**

<table>
<thead>
<tr>
<th>Label</th>
<th>Operation Field</th>
<th>Variable Field</th>
</tr>
</thead>
<tbody>
<tr>
<td>Optional</td>
<td>DEC</td>
<td>One or more decimal, octal, or hexadecimal constants (separated by commas)</td>
</tr>
</tbody>
</table>

This statement is provided for compatibility with other assemblers. Each constant in the variable field is evaluated as a decimal constant, converted into one or more binary words, and loaded starting at the current location count. All formats accepted by the DATA statement may be used with DEC except the repeated constant format (3X'12AB'). (See Table 4-2.) Hexadecimal and octal constants are interpreted as single precision fixed point.

**Examples:**

```
05302:  000020  (0040)       DEC  16
05303:  000003  (0041)       DEC  3, 10, 2, X'1A', $F, '123
05304:  000012
05305:  000002
05306:  000032
05307:  000017
05310:  000123
```
DBP (Set Double Precision Constant)

<table>
<thead>
<tr>
<th>Label</th>
<th>Operation Field</th>
<th>Variable Field</th>
</tr>
</thead>
<tbody>
<tr>
<td>Optional</td>
<td>DBP</td>
<td>One or more decimal, octal, or hexadecimal constants (separated by commas)</td>
</tr>
</tbody>
</table>

This statement provides compatibility with other assemblers. Each constant in the variable field is evaluated as a decimal constant, converted to double precision binary format, and loaded in consecutive memory cells starting at the current location count. The format of each expression determines whether the result will be fixed or floating point. For fixed point quantities, the assembler is forced to assign the first word to an even location count. (If the current count is odd, it is skipped.) The repeating constant format of the DATA statement is not permitted.

Examples:

05314: 000000 (0043)         DBP 16        FIXED POINT
05315: 000000
05316: 000000 (0044)
05317: 000000
05320: 000000
05321: 040316
05322: 000000
05323: 000000
05324: 000000
05325: 000000
05326: 000000
05327: 000123
05330: 040316 (0045)         DBP 1.23       FLOATING POINT
05331: 134121
05332: 127075 (0046)
05333: 114301
05334: 000000
05335: 000000
05336: 040740
05337: 000000

4-24
**OCT (Set Octal Constant)**

<table>
<thead>
<tr>
<th>Label</th>
<th>Operation Field</th>
<th>Variable Field</th>
</tr>
</thead>
<tbody>
<tr>
<td>Optional</td>
<td>OCT</td>
<td>One or more octal constants (separated by commas)</td>
</tr>
</tbody>
</table>

This statement is provided for compatibility with other assemblers. Each constant in the variable field is evaluated as an octal constant, converted to single precision fixed point binary, and loaded at the current location count. Only the following constant forms are allowed.

**Examples:**

<table>
<thead>
<tr>
<th>Label</th>
<th>Operation Field</th>
<th>Variable Field</th>
</tr>
</thead>
<tbody>
<tr>
<td>05340:</td>
<td>000012 (0047)</td>
<td>OCT 12</td>
</tr>
<tr>
<td>05341:</td>
<td>070017 (0048)</td>
<td>OCT 70017, 321, 677</td>
</tr>
<tr>
<td>05342:</td>
<td>000321</td>
<td></td>
</tr>
<tr>
<td>05343:</td>
<td>000677</td>
<td></td>
</tr>
<tr>
<td>05344:</td>
<td>177777 (0049)</td>
<td>OCT 177777, -1, -13, +177, +177</td>
</tr>
<tr>
<td>05345:</td>
<td>177777</td>
<td></td>
</tr>
<tr>
<td>05346:</td>
<td>177765</td>
<td></td>
</tr>
<tr>
<td>05347:</td>
<td>000177</td>
<td></td>
</tr>
<tr>
<td>05348:</td>
<td>000177</td>
<td></td>
</tr>
</tbody>
</table>
**HEX (Set Hexadecimal Constants)**

<table>
<thead>
<tr>
<th>Label</th>
<th>Operation Field</th>
<th>Variable Field</th>
</tr>
</thead>
<tbody>
<tr>
<td>Optional</td>
<td>HEX</td>
<td>One or more hexadecimal constants (separated by commas)</td>
</tr>
</tbody>
</table>

This pseudo-op is provided for compatibility with other assemblers. It converts the hex constants within the variable field to single precision binary values and loads them in consecutive locations starting at the current location count. Only the following constant forms are allowed.

**Examples:**

<table>
<thead>
<tr>
<th>Address</th>
<th>Contents</th>
<th>Operation</th>
<th>Field</th>
</tr>
</thead>
<tbody>
<tr>
<td>05351</td>
<td>011253 (0050)</td>
<td>HEX</td>
<td>12AB</td>
</tr>
<tr>
<td>05352</td>
<td>000017 (0051)</td>
<td>HEX</td>
<td>F, $F, -FFF1, -$FFF1</td>
</tr>
<tr>
<td>05353</td>
<td>000017</td>
<td></td>
<td></td>
</tr>
<tr>
<td>05354</td>
<td>000017</td>
<td></td>
<td></td>
</tr>
<tr>
<td>05355</td>
<td>000017</td>
<td></td>
<td></td>
</tr>
<tr>
<td>05356</td>
<td>000017</td>
<td></td>
<td></td>
</tr>
<tr>
<td>05357</td>
<td>011253 (0052)</td>
<td>HEX</td>
<td>$12AB, +12AB, +$12AB, +12AB</td>
</tr>
<tr>
<td>05360</td>
<td>011253</td>
<td></td>
<td></td>
</tr>
<tr>
<td>05361</td>
<td>011253</td>
<td></td>
<td></td>
</tr>
<tr>
<td>05362</td>
<td>011253</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
VFD (Define Variable Fields)

<table>
<thead>
<tr>
<th>Label</th>
<th>Operation Field</th>
<th>Variable Field</th>
</tr>
</thead>
<tbody>
<tr>
<td>Optional</td>
<td>VFD</td>
<td>One or more subfields of the form:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Field Size, Value</td>
</tr>
</tbody>
</table>

This statement permits 16-bit data words to be formed in subfields of varying length by pairs of constants (field size, value) in the variable field. The first constant of each pair specifies a number of adjacent bits, starting at the most significant end of the 16-bit word. The second constant of a pair is the value to be loaded. Subsequent field size value pairs load less significant subfields of the 16-bit word. For any pair, if a value exceeds the specified field size, the more significant overflow bits are exclusive OR'ed with the subfield to the left. (No error message is generated.) If the entire word is not specified, the least significant end is filled with zeroes. An error message is printed if the assembler attempts to load more than 16 bits.

Examples:

VFD  8,C'A'-2, 8, 0
VFD  SZ/4, X, SZ/4, Y, SZ/4, Z, SZ/4, X'F'
BCI (Define ASCII String)

<table>
<thead>
<tr>
<th>Label</th>
<th>Operation Field</th>
<th>Variable Field</th>
</tr>
</thead>
</table>
| Optional | BCI             | 'STRING; (where ' is any non-zero, non-digit delimiter) or 
|         |                 | #,STRING (where # is the number of character pairs) |

This statement loads ASCII character strings by packing the specified ASCII characters two per word, starting with the most significant 8 bits. Assembled words are loaded starting at the current location count.

In the first format, the string is delimited by any character other than zero or a digit:

```
05365: 140702 (0056)   BCI  'AB'
05366: 140702 (0057)   BCI  '/ABC3450X/
05367: 141663
05370: 132265
05371: 130330
```

If an odd number of characters is specified, the least significant half of the last word is padded with zeroes.

In the second format, the character string is preceded by a word count (the number of characters divided by 2 and rounded up):

```
05372: 120240 (0058)   BCI  *   *   (12 SPACES)
05373: 120240
05374: 120240
05375: 120240
05376: 120240
05377: 120240
05400: 140702 (0059)   BCI  1,AB
05401: 140702 (0060)   BCI  4,ABC2340X
05402: 141662
05403: 131664
05404: 130330
05405: 120240 (0061)   BCI  6,   (12 SPACES)
05406: 120240
05407: 120240
05410: 120240
05411: 120240
05412: 120240
```

4-28
DAC (Local Address Definition)

<table>
<thead>
<tr>
<th>Label</th>
<th>Operation Field</th>
<th>Variable Field</th>
</tr>
</thead>
<tbody>
<tr>
<td>Optional</td>
<td>DAC</td>
<td>Address Expression</td>
</tr>
<tr>
<td>or</td>
<td>DAC* (indirect addressing)</td>
<td>Address Expression, 1 (Indexing)</td>
</tr>
</tbody>
</table>

This statement loads the current location with an address word consisting of up to 15 address bits, with optional indexing and indirect address bits. The address is specified by the expression in the variable field. Indexing and indirect addressing may be specified symbolically as in memory reference instructions (* and, 1). Address words formed by DAC are subject to the effects of the EXD pseudo-operation and the E16S, E32S, and E32R instructions. If relocatable mode is in effect the loader performs relocation during loading.

**Examples:**

```
05413: 00.77777 (0062)       DAC ALPHA
05414: 00 05173 (0063)       DAC AP3+3
05415: 20 05227 (0064) X123  DAC XYZ.1
05416: 40.00003 (0065)       DAC* K31/2+3
05417: 60.00020 (0066)       DAC* '20.1
05420: 00.00000 (0067) SUB1   DAC ** (TYPICAL SUBROUTINE ENTRY)
```

In the assembler, the DAC pseudo-op generates a 16-bit constant. The loader truncates this constant to 14 bits if in the LXD mode, 15 bits if in the EXD mode, or does not truncate it if absolute. The loader merges the index bit with the address constant. It merges the indexing bit in normal addressing mode, and ignores it in extended (E32S) mode.
XAC (External Address Definition)

<table>
<thead>
<tr>
<th>Label</th>
<th>Operation Field</th>
<th>Variable Field</th>
</tr>
</thead>
<tbody>
<tr>
<td>Optional</td>
<td>XAC</td>
<td>External variable</td>
</tr>
<tr>
<td>or</td>
<td>XAC* (Indirect addressing)</td>
<td>External Variable, i (Indexing)</td>
</tr>
</tbody>
</table>

Generates the same type of data word as DAC. However, the variable field is interpreted as an external variable that has no relation to, or conflict with, an internal variable of the same name.

Examples:

05421: 00 00000 (0068)        XAC  FLAGS
05422: 00 00000 (0069) T20    XAC  R3*
05423: 00 00000 (0070)        XAC  T$1, i
05424: 40 00000 (0071)        XAC*  T$1
05425: 40 00000 (0072)        XAC*  T$1, i
***(Dummy Memory Reference Instruction)**

<table>
<thead>
<tr>
<th>Label</th>
<th>Operation Field</th>
<th>Variable Field</th>
</tr>
</thead>
<tbody>
<tr>
<td>Optional</td>
<td>***</td>
<td>Address Expression</td>
</tr>
<tr>
<td>or</td>
<td></td>
<td>or</td>
</tr>
<tr>
<td>or</td>
<td>*** (Indirect</td>
<td>Address Expression, l</td>
</tr>
<tr>
<td></td>
<td>Addressing)</td>
<td>(Indexing)</td>
</tr>
</tbody>
</table>

Causes the assembler to create a dummy memory referencing instruction with zeroes in the op-code field. Indirect addressing is indicated by an asterisk, as usual (resulting in a four-asterisk operation field) and indexing may be specified. This statement is used when the op-code is to be calculated and placed in the op-code field at run time, prior to execution.
VARIABLE (SYMBOL) DEFINING PSEUDO-OPERATIONS

Variables used as address symbols are usually defined when they appear in the label field of an instruction or pseudo operation statement. Symbols so defined are given the numerical value of the statement's location count. The EQU and SET statements make it possible to equate symbols to any numerical value, even ones that lie outside the range of addresses in a program.

EQU (Define Variable),
SET (Redefine Variables)

<table>
<thead>
<tr>
<th>Label</th>
<th>Operation Field</th>
<th>Variable Field</th>
</tr>
</thead>
<tbody>
<tr>
<td>Format A</td>
<td>Contains a variable</td>
<td>EQU or SET</td>
</tr>
<tr>
<td>Format B</td>
<td>Blank</td>
<td>EQU or SET</td>
</tr>
</tbody>
</table>

In format A, the variable in the label field is equated to the address expression. Any variables used in the address expression must already be defined:

```
000003 (0073) I       EQU 3
077777 (0074) ALPHA    SET 32767
177777 (0075) PRIME    EQU '177777
077773 (0076) BETA     SET ALPHA-4
```

In format B, symbols are assigned numerical values by equality expressions in the address field. One or more equality expressions can be used, separated by commas:

```
000003 (0077)         EQU I=3
034777 (0078)         EQU J=39FF, K='7777, L='AZ'
000022 (0079)         SET K=18
035376 (0080)         SET K=J+$FF
```

Formats A and B can be combined in a single statement:

```
000003 (0081) I       SET 3, J=39FF, K='7777
```

EQU and SET perform the same functions; however, a variable defined by EQU may not be redefined, while a variable...
once defined by SET may be redefined by subsequent SET statements without causing an error message.

Examples:

```assembly
  00020  00021  00022  00023  00024  00025  00026  00027  00028  00029  00030  00031  00032  00033  00034
CHAN1  STRTADR EQU  EQU  '20  EQU  BUF1  DMA CHANNEL 1  STARTING ADDRESS I/O TRANSFER
       SET  SET
  00036  00037  00038  00039  00040  00041  00042  00043  00044  00045  00046  00047  00048  00049  00050
   012  013  014  015  016  017  018  019  020  021  022  023  024  025  026  027  END
```

BUF1  00015  0004  0019
BUF2  00041  0013  0020
CHAN1 00020  0003  0009  0015
STRTADR 00041  0004  0008  0013  0014
STORAGE ALLOCATION PSEUDO-OPERATIONS

BSS (Block Starting with Symbol),
BES (Block Ending with Symbol),
BSZ (Block Set to Zeroes)

<table>
<thead>
<tr>
<th>Label</th>
<th>Operation Field</th>
<th>Variable Field</th>
</tr>
</thead>
<tbody>
<tr>
<td>Optional</td>
<td>BSS, BES or BSZ</td>
<td>Expression that specifies number of words to be allocated</td>
</tr>
</tbody>
</table>

These statements allocate a block of words of the size specified in the variable field, starting at the current location count. If there is a label, it is assigned to the first word of the block (BSS and BSZ) or to the last word of the block +1 (BES). For BSZ, all words within the block are set to zeroes.

Examples:

L1   BSS   20
     BSS   (I+3)/2

T1   BES   40
     BES   N*3-2

Z1   BSZ   100
     BSZ   (AB-2)*3
## SETC (Set Common Base Address)

<table>
<thead>
<tr>
<th>Label</th>
<th>Operation Field</th>
<th>Variable Field</th>
</tr>
</thead>
<tbody>
<tr>
<td>Not used</td>
<td>SETC</td>
<td>Address expression</td>
</tr>
</tbody>
</table>

The address expression specifies a location near the top of memory to be used by the loader as the COMMON base (the highest location in a pool of common items). In systems with over 16K of memory, the expression specifies an address in the current 16K of memory. Variables in the address expression must be defined and the result must be absolute.

### Examples:
- `SETC '17770`
- `SETC END-8`
COMN (Define Common Items)

<table>
<thead>
<tr>
<th>Label</th>
<th>Operation Field</th>
<th>Variable Field</th>
</tr>
</thead>
<tbody>
<tr>
<td>Optional</td>
<td>COMN</td>
<td>One or more variables</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(separated by commas)</td>
</tr>
</tbody>
</table>

This statement loads common variables in the COMMON area at the top of memory. Each of the variables in a COMN statement is assigned an address starting with a common base selected by the loader or set by a SETC statement. Variables are assigned addresses in the order they appear in the variable field, and addresses are assigned in decreasing order. The loader keeps track of the last COMMON address assigned, and in subsequent COMN statements continues to assign lower COMMON locations in sequence, until another SETC statement is encountered.

Examples:

```
0001: *PROGRAM A
0002: REL
0003: SETC '13777
013777 0004: COMN AA, AB, AC
013776
013775

0000: 02 00004 (0005) LDA =1
0001: 04 13777 (0006) STA AA
0002: 12 13775 (0007) IRS AC
0003: 000000 (0008) HLT
000004 (0009) END

00004: 000001

AA 013777, 0004 0006
AB 013776, 0004
AC 013775, 0004 0007
```
(0001) *PROGRAM B
(0002) *
(0003) REL
(0004) SETC '13777' SET COMMON BASE
013777 (0005) COMN BR, BB, BC, BD ASSIGN FOUR LOCATIONS IN COMMON
013775
013774

00000: 02 13777 (0006) LDA EA CHECK PROGRAM FLAG
00001: 100040 (0007) SIZ UPDATE COUNTER BD
00002: 12 13774 (0008) IRS ED
00003: 02 00006 (0009) LDA =1
00004: 04 13776 (0010) STA EB SET FLAG BB
00005 (0012) END

00006: 000001

BA 013777/ 0005 0006
BB 012776/ 0005 0010
BC 013775/ 0005
BD 013774/ 0005 0008

C PROGRAM C (FORTRAN)
C
C DECLARE COMMON IN THE REVERSE ORDER OF PMA EXAMPLES
COMMON CF,CE,CD,CC,CB,CA
C CHECK FLAG OF PROGRAM B
IF (CE) 10,20,10
000000 JMPE 000000
000001 LNK 000000
000002 CALL L#22
000003 DCC CB +
C UPDATE COUNTER CE
000004 SNZ
000005 JMPE .20
10 CF=CE+1 C SET FLAG OF PROGRAM C
000006 CALL L#22
000007 DCC CE +
000008 CALL A#22
000009 DAC ="040300"
000010 CALL H#22
000011 DAC ="040300"
000012 DCC CF +
20 CF=1
000013 LDA ="000001"
000014 CALL O#12
000015 CALL H#22
000016 DAC CF +
STOP
000017 CALL P#HT
000018 DAC ="000001"
000019 JMF 000017
END
000020 DAC .10
000021 DFC .20
000022 OCT 040300
000023 OCT 000000
000024 LNK ="000001"
PROGRAM LINKING PSEUDO-OPERATIONS

This group of statements coordinates the interaction of the assembler and loader in resolving address references between main programs and external subroutines. EXT and CALL are used in main programs to identify external names. ENT and SUBR are used in subroutines to tell the loader what names appear in the subroutine.

EXT (Flag External References)

<table>
<thead>
<tr>
<th>Label</th>
<th>Operation Field</th>
<th>Variable Field</th>
</tr>
</thead>
<tbody>
<tr>
<td>Optional EXT</td>
<td>One or more external entry point names</td>
<td></td>
</tr>
</tbody>
</table>

The names appearing in the variable field of this statement are flagged as being external references. Whenever other statements in the main program make reference to one of these names, a special block of object text is generated that notifies the loader that it must fill in the address properly. (The assembler fills the address fields with zeroes.) If the loader encounters any EXT statements while loading a main program, it will print the MR message after loading is complete, to notify the operator that the external subroutines containing the names must be loaded also.

Names defined by the EXT pseudo-op are unique only in the first 6 characters (Loader restriction) and should not appear in a label field internal to the program.

**Examples:**

```
LDA TST2
.
.
EXT TST2
```

If TST2 is a location in an external subroutine, the EXT statement is required. Otherwise the loader will be unable to resolve the address reference.
CALL (External Subroutine Reference)

<table>
<thead>
<tr>
<th>Label</th>
<th>Operation Field</th>
<th>Variable Field</th>
</tr>
</thead>
<tbody>
<tr>
<td>Optional</td>
<td>CALL</td>
<td>External Entry Point</td>
</tr>
<tr>
<td></td>
<td>(* for indirect addressing is optional)</td>
<td>(.1 for indexing is optional)</td>
</tr>
</tbody>
</table>

This statement generates object coding that has the same effect on the loader as a JST to the name specified in the variable field followed by an EXT statement that defines that name as external. For example, the statement CALL TST1 generates object coding that is equivalent to the statements:

```
JST TST1
EXT TST1
```

The variable field must contain a single variable (not an expression) of up to 6 characters.

**Examples:**

```
CALL SIN

B3 CALL F$IO

CALL* TLIST

CALL TABLE6,1

CALL* ARRAY,1
```
SUBR, ENT (Define Entry Points)

These pseudo-operations are identical in effect. They are used in external subroutines to link subroutine entry points to external names used in CALL, XAC, or EXT statements in main programs. Both mnemonics are provided for compatibility with other assemblers. The form is:

<table>
<thead>
<tr>
<th>Label</th>
<th>Operation Field</th>
<th>Variable Field</th>
</tr>
</thead>
<tbody>
<tr>
<td>Optional</td>
<td>SUBR or ENT</td>
<td>Extname</td>
</tr>
<tr>
<td></td>
<td></td>
<td>or</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Extname, Entryname</td>
</tr>
</tbody>
</table>

where Extname is the external name used in the main program, and Entryname is the name of the entry point in the subroutine, if different from Extname.

Examples:

Main
Program

CALL TST1
.
.
.
.

----------------------------------------
External Subroutine

SUBR TST1
.
.
.
TST1 DAC **
.
.
.
JMP * TST1
END

This is a simple case where external name TST1 is linked by a SUBR statement to entry point TST1 of the external subroutine. When the main program uses a different external name, the SUBR statement can equate names as follows:
Main Program

CALL MAINT1

EXTERNAL Subroutine

SUBR MAINT1,TST1

TST1 DAC **

JMP * TST1

END

The name MAINT1 is equated to the actual entry point TST1 by the SUBR statement.

ENT statements have the same effect as SUBR statements but usually identify entry points or locations other than the main subroutine entry point. For example:

Main Program

CALL MAINT1,TST1

LDA TST2

JMP TST3

EXT TST2

EXT TST3
External Subroutine

ENT TST1, TST2
ENT TST3

. .

TST1 DAC **

. .

TST2 OCT '77

. .

TST3 LDA XYZ

. .

Here, the main program refers to two locations in the external subroutine, TST2 and TST3. The EXT statements in the main program notify the loader that the names are external. The ENT statements in the subroutine notify the loader that the subroutine contains those names.

ENT statements also permit the main program to use different names from those used in the subroutine; for example,

Main Program

JMP TEST2

. .

EXT TEST2

. .

-----------------------------------------------

External Subroutine

ENT TEST2, TST2

. .

TST2 LDA XYZ

. .

4-42
As many SUBR or ENT statements may be used as are needed, and the statements may appear anywhere within the subroutine. However, only the object code following the pseudo-operation will be loaded. Thus several subroutines can be packed in a single tape or file, and only the ones that are specified by SUBR or ENT statements will be loaded.

Since the loader restricts external names to 6 characters maximum, only the first 6 characters of any name in the variable field of the ENT or SUBR statement are used as the name internal to the main program.
CONDITIONAL ASSEMBLY PSEUDO OPERATIONS

IF (Conditional Statement)

<table>
<thead>
<tr>
<th>Label</th>
<th>Operation Field</th>
<th>Variable Field</th>
</tr>
</thead>
<tbody>
<tr>
<td>Optional</td>
<td>IF</td>
<td>(Expression)(Statement) : (Statement): (etc.)</td>
</tr>
</tbody>
</table>

The variable field consists of an expression followed by one or more instruction or pseudo-operation statements separated by colons. If the expression is true (has a non-zero result) the rest of the line is assembled. Otherwise the rest of the line is ignored and the next line is processed. The variable field of the IF statement must not be continued into the following line, because the skip-if-false condition proceeds to the next physical rather than logical line.

Examples:

IF FLAG SET FLAG=0 ; GO TO A24
IF (COUNT .LT. MAX) SET COUNT = COUNT + 1
IF (CONTROL .EQ. 134) GO TO FIXC
IF (N .NE. M) LDA N : AOA ; STA M
IF (OPTION .AND. '01000 .EQ. 1) GO TO S130
IFM (Continue Assembly if Minus)
IFP (Continue Assembly if Plus)
IFZ (Continue Assembly if Zero)
IFN (Continue Assembly if Not Zero)

This group of pseudo-operations is provided for compatibility with other assemblers.

<table>
<thead>
<tr>
<th>Label</th>
<th>Operation Field</th>
<th>Variable Field</th>
</tr>
</thead>
<tbody>
<tr>
<td>Optional</td>
<td>IFM</td>
<td>Expression</td>
</tr>
<tr>
<td></td>
<td>IFP</td>
<td></td>
</tr>
<tr>
<td></td>
<td>IFZ</td>
<td></td>
</tr>
<tr>
<td></td>
<td>IFN</td>
<td></td>
</tr>
</tbody>
</table>

The expression in the variable field is evaluated. If the result matches the IF condition, assembly proceeds normally. Otherwise, the assembler ignores all subsequent statements until an ENDC statement is reached.

For every IFx statement there must be a matching ENDC statement. IFx and ENDC pairs may be nested within each other. The nesting depth count is checked even in sections of code that are being skipped by a previous IFx statement.

Examples:

IFP B20   (continue assembly if B20 is \( \geq 0 \))
IFM (I+3-24) (continue assembly if expression < 0)
IFZ (ALPHA-6)(continue assembly if expression = 0)
IFN X24-1 (continue assembly if expression \( \neq 0 \))
ENDC (End Conditional Assembly Area)

<table>
<thead>
<tr>
<th>Label</th>
<th>Operation Field</th>
<th>Variable Field</th>
</tr>
</thead>
<tbody>
<tr>
<td>Not Used</td>
<td>ENDC</td>
<td>Not Used</td>
</tr>
</tbody>
</table>

Defines the end of a conditional assembly area started by an IFP, IFM, IFZ, or IFN statement. Every IFx statement must have a matching ENDC.

ELSE (Reverse Conditional Assembly)

<table>
<thead>
<tr>
<th>Label</th>
<th>Operation Field</th>
<th>Variable Field</th>
</tr>
</thead>
<tbody>
<tr>
<td>Not Used</td>
<td>ELSE</td>
<td>Not Used</td>
</tr>
</tbody>
</table>

Reverses the condition set up by an IFx statement until the matching ENDC statement is reached. If the IFx condition inhibited assembly, the ELSE statement enables assembly, and vice versa. ELSE statements that lie within the bounds of other IFx-ENDC pairs nested within the conditional assembly area are ignored.

Examples:

```
(0067) *-----TEST OF ELSE (WITH OLD-STYLE IFs).
(0068)   IFP   FIVE
(0070)   ELSE
(0072)   ENDC
(0073)   IFP   MTWO
(0075)   ELSE
(0077)   ENDC
(0078)   IFP   MTWO
(0080)   IFP   FIVE
(0082)   ELSE
(0084)   ENDC
(0086)   ELSE
01002: 02.00007 (0087)   LDA   7
(0088)   ENDC
```
FAIL (force Error Message)

<table>
<thead>
<tr>
<th>Label</th>
<th>Operation Field</th>
<th>Variable Field</th>
</tr>
</thead>
<tbody>
<tr>
<td>Optional</td>
<td>FAIL</td>
<td>Not used</td>
</tr>
</tbody>
</table>

The assembler responds to a FAIL statement by printing the error message "F". This notifies the operator of a logical or range error, for example within the range of a conditional IFx statement, that has caused the assembly to proceed to an undesirable location.
SECTION 5
MACRO FACILITY

The macro feature of this assembler enables the programmer to define functions that can be expressed in easily interpreted English (or other) language statements, such as:

TRANSFER DATA TO DAC
TURN ON VALVE 312

Once a macro function has been defined, it can be called for use over and over again within a program. New argument values (DATA, DAC, ON 312) can be provided with every call. Dummy words (TO, VALVE) can be used to increase intelligibility. Such words can be identified during macro definition so that they will not be treated as arguments when they appear in a macro call.

After a set of macros has been defined by a system-level programmer, a specialist in a particular application field can formulate macro calls to solve his application problems, without becoming involved in the details of assembly language programming.

Macros are defined by the MAC and ENDM pseudo-operations. These and other features of macro definition, listing, and assembly are discussed in detail in this section.
MACRO DEFINITIONS AND CALLS

Two pseudo-operations are provided for macro definition: The MAC and ENDM statements.

MAC (Begin Macro Definition)

<table>
<thead>
<tr>
<th>Label Field</th>
<th>Operation Field</th>
<th>Variable Field</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name of macro (to be used in operation field of macro calls)</td>
<td>MAC</td>
<td>Optional dummy words and/or argument identifiers (see text) separated by commas.</td>
</tr>
</tbody>
</table>

A MAC statement begins the definition of a macro named by the label field. The name is formed in the same way as any variable or label. Following the MAC statement are the statements that make up the macro definition; for example:

TRANSFER MAC

LDA <1>

STA <2>

ENDM

The integers enclosed in angle brackets are argument references. During assembly they are replaced by argument values specified in a macro call. Optional dummy words ("noise words") and argument identifiers ("positional noise words") are described later.

Macro definitions may contain macro calls to any depth, but macro definitions themselves cannot be nested.
ENDM (End Macro Definition)

The macro definition must be concluded by an ENDM statement:

<table>
<thead>
<tr>
<th>Label</th>
<th>Operation Field</th>
<th>Variable Field</th>
</tr>
</thead>
<tbody>
<tr>
<td>Optional</td>
<td>ENDM</td>
<td>Ignored</td>
</tr>
</tbody>
</table>

This statement terminates assembly of the macro.

Argument References

Argument references (in angle brackets) may be specified in any field of a statement within a macro definition. The number within the angle brackets may be a variable or an expression, provided all variables within the expression are previously defined as absolute integer values at the time the macro is called.

Example: LDA <I> + <J-I+1>

Argument references may be nested to any desired depth.

Example: <I + <3 - <J> -1>

Arguments <I> and up are replaced by argument values from the variable field of a macro call during assembly.

Argument <0> is replaced by the label field of the macro call during assembly. The label of the macro call is not automatically assigned.

Example: <0> LDA <3> - 1

Macro Calls

A macro call is a special type of statement that uses the name of a defined macro in the operation field:

<table>
<thead>
<tr>
<th>Label</th>
<th>Operation Field</th>
<th>Variable Field</th>
</tr>
</thead>
<tbody>
<tr>
<td>Optional</td>
<td>Name of User-Defined or Library Macro</td>
<td>Argument value expressions, plus optional dummy words or argument identifiers, separated by commas or blanks</td>
</tr>
</tbody>
</table>

For each macro call, the assembler enters the in-line code of the defined macro starting at the current location. Argument references are replaced by argument values from the variable field.
User-defined macros must be defined in source statements preceding the macro call.

Here is a a typical call to the TRANSFER macro defined above:

TRANSFER ARG1, '1770

Argument Values

The variable field of a macro call usually contains one or more expressions to be interpreted as argument values. An argument value expression starts with the first non-space character of the variable field and continues until a terminating comma or space occurs. (The comma or space is not considered part of the argument expression.)

Argument Substitution

During assembly of a macro call, the assembler substitutes the argument values in the macro call variable field for the argument references in the macro definition. Argument expressions are matched to argument references in numerical order from left to right:

<table>
<thead>
<tr>
<th>Variable Field</th>
<th>Argument &lt;1&gt;</th>
<th>Argument &lt;2&gt;</th>
<th>Argument &lt;3&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>A</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>A+3</td>
<td>A+3</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>X,Y-1,Z*A-1</td>
<td>X</td>
<td>Y-1</td>
<td>Z*A-1</td>
</tr>
<tr>
<td>X,B-C (Z3*X2)</td>
<td>X</td>
<td>B-C</td>
<td>Z3*X2</td>
</tr>
<tr>
<td>(A,B-1), C</td>
<td>A,B-1</td>
<td>C</td>
<td></td>
</tr>
<tr>
<td>(X,Y,(Z1+Z2),3)</td>
<td>X,Y,Z1+Z2,3</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

The first expression in the macro call is assigned as argument 1, the second as argument 2, and so on. In the following call to the TRANSFER macro -

TRANSFER ARG1, '1770

The variable ARG1 is argument 1 and the constant '1770 is argument 2. Thus, the macro example is assembled as:

LDA ARG1

STA '1770

Arguments that are not assigned values in a macro call are set to zero by the assembler.
Argument Values in Parentheses

Argument value expressions may be enclosed in parentheses to permit the use of commas, spaces, or string delimiters within a single argument. (The outside parentheses are not included as part of the argument expression.) One use of this is in forming sub-lists of arguments for macro calls nested within a given macro definition.

Examples:

MACRO
DEFINITION
CONTAINING
CALL TO "TRANSFER"
MACRO

WAIT
MAC
IRS <1>
JMP * -1
TRANSFER <2>
ENDM

CALL TO WAIT MACRO

WAIT 100, (ARG1, ARG2)

ASSEMBLED
AS

IRIS 100
JMP * -1
LDA ARG1
STA ARG2

Dummy Words

An ordinary macro like:

TRANSFER ARG1, ARG2

is simple, but cryptic. A few extra words in the variable field of the macro call can improve the intelligibility greatly as in:

TRANSFER ARG1 TO ARG2
TRANSFER DATA TO PRINTER
TRANSFER MESSAGE TO TTY
TRANSFER FROM CONSOLE TO DISPLAY
and so on. These macro calls are made self-documenting by a combination of meaningful argument symbols (DATA, MESSAGE, PRINTER etc.) plus "dummy" words, such as TO, FROM. Dummy words are ignored by the assembler (ie., not mistaken for argument symbols).

Dummy words applicable to a given macro are assigned in the variable field of the MAC statement that starts the macro definition, as in:

```
TRANSFER MAC TO
```

In this statement, TO is defined as a dummy word. In any subsequent call to this macro, the assembler ignores the word TO (does not mistakenly assume it to be a symbol to be substituted for an argument). All other expressions in the variable field are interpreted as arguments (proceeding in numerical argument order from left to right) and substituted for the argument numbers in the macro definition statements as usual. When TRANSFER macro is called by a statement

```
TRANSFER ALPHA TO '7770
```

the assembler ignores the TO and assembles the macro as if the call statement were TRANSFER ALPHA, '7770.

A dummy word string can be any number of ASCII characters (letters, numerals, period and $ sign). Any number of dummy word strings may be used in a macro call, separated by commas. If the first character of a dummy word string is an open parenthesis, all characters (including spaces and commas) up to the closing parenthesis are considered part of the same string. (The surrounding parentheses are not included).

Here are some possible variations of the TRANSFER macro:

<table>
<thead>
<tr>
<th>MAC Statement</th>
<th>Macro Call</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRANSFER MAC DATA, FROM, TO</td>
<td>TRANSFER DATA FROM ALPHA TO '7770</td>
</tr>
<tr>
<td>TRANSFER MAC VOLTS, TO, DIG OUT</td>
<td>TRANSFER 3.22 VOLTS TO DIG OUT 14</td>
</tr>
<tr>
<td>TRANSFER MAC COUNTS, TO, PULSER</td>
<td>TRANSFER 3374 COUNTS TO PULSER $FF</td>
</tr>
</tbody>
</table>

Arguments are underlined.
Other examples of typical macro calls using dummy words:

INPUT S1, M1, S2, S3 AND M6
ADJUST K2 BY K3, T4 BY K3 AND T5 BY T1
MOVE 3 WORDS FROM X31 TO Z21
SUM X1, X2, X3 AND X4
DISPLAY ALPHA
CONNECT 7.0 VOLTS TO PIN 5 ON CONNECTOR 1

Argument Identifiers

The self-documenting effect of dummy words improves the intelligibility of macro calls, but the programmer must be careful to enter values for arguments in the proper order. Argument identifiers increase the format flexibility of macro calls by associating a particular argument number with a specific dummy word, regardless of order. For example, identifiers can be defined so that argument 1 follows the dummy word "TO", and argument 2 follows "FROM", regardless of the order in which TO and FROM appear in the macro call.

Argument identifiers, like dummy words, are assigned in the variable field of a MAC statement that introduces a macro definition. An argument identifier word consists of a dummy word enclosed in parentheses and equated to an argument number:

TRANSFER MAC (FROM) = 1, (TO) = 2
.
.
ENDM

When a call to the macro uses a defined argument identifier in its variable field, the first non-dummy expression immediately following the identifier is taken as the value of the argument:

TRANSFER FROM ALPHA TO BETA
TRANSFER TO BETA FROM ALPHA

Both of these calls have the same effect: the expression following the dummy word FROM is taken as argument <1>, and the expression following TO is taken as argument <2>.
Argument identifiers and dummy words may be used together in the same macro. Ordinary dummy words are ignored, as usual.

Arguments that are not associated with identifier words receive values in the usual positional priority - the first non-dummy word is taken as the value for the first unspecified argument, and so on. Example:

Macro Definition:  MASK MAC (BY)=2, (TO)=3, MASK, TRANSFER, AND

    LDA <1>
    ANA <2>
    STA <3>
    ENDM

Macro Call:  MASK INPUT BY =7 AND TRANSFER TO BUFF1

Here, argument 2 is =7 and argument 3 is BUFF1, as located by identifier words BY and TO. Argument 1 is assigned the value of the expression INPUT (the only other non-dummy word in the variable field).

Assembler Attribute References

Certain useful attributes of a macro can be specified by a number preceded by the pound character (#). The following assembler attributes are presently available to the macro programmer:

    #1 = Current Macro Call Number.
    #2 = Number of Arguments in Current Macro Call.

(others may be assigned later)

The attribute number may be a variable, or an expression within parentheses, as long as such variables are previously defined as absolute integer values. Attribute references are evaluated as absolute integer values.

Examples:  #3
            #XYZ
            #(I+2)
            # <3>

5-8
Local References Within Macros

Local labels can be assigned within a macro definition by using the ampersand character (&) as the first character of the label. Local labels do not conflict with labels outside of the macro. The ampersand is replaced by a 4-digit macro call number, thereby assuring uniqueness of the label regardless of the macro's environment. Use of the "&" outside of a macro will result in the substitution of 4 zeros.

<table>
<thead>
<tr>
<th>Examples:</th>
<th>Local Label</th>
<th>Evaluated As</th>
<th>In Macro Call</th>
</tr>
</thead>
<tbody>
<tr>
<td>&amp;ABC</td>
<td>0002ABC</td>
<td></td>
<td>0002</td>
</tr>
<tr>
<td>&amp;X3A</td>
<td>1739X3A</td>
<td></td>
<td>1739</td>
</tr>
</tbody>
</table>
MACRO LISTING AND ASSEMBLY CONTROL

Three levels of listing detail for macro calls are provided. The default condition is NLSM, which causes only the Macro call statement to be processed (no detailed output). These pseudo-ops are global in that they remain in effect until a new macro listing control pseudo-op is specified.

The BACK (TO) and SAY statements control assembly of macros.

LSTM (List Macro Expansions)

<table>
<thead>
<tr>
<th>Label</th>
<th>Operation Field</th>
<th>Variable Field</th>
</tr>
</thead>
<tbody>
<tr>
<td>Optional</td>
<td>LSTM</td>
<td>Not used</td>
</tr>
</tbody>
</table>

Directs assembler to list macro call statements and all lines generated by expansion of the macro, including code or data values.

Example:

```
(0001) *TRANSFER MACRO EXAMPLE
(0002) *
(0003) 
010000 (0004) START ORG '1000
(0005) TRANSFER MAC TO
(0006) LDA <1>
(0007) OTA <2>
(0008) ENDM
01000: 02 0100P (ML01) LDA DATA
01001: 171777 (ML01) OTA DAC
01002: 000047 (0010) DATA DEC 39
001777 (0011) DAC SET '1777
001003 (0012) END
```
**LSMD (List Macro Expansions - Data Only)**

<table>
<thead>
<tr>
<th>Label</th>
<th>Operation Field</th>
<th>Variable Field</th>
</tr>
</thead>
<tbody>
<tr>
<td>Not used</td>
<td>LSMD</td>
<td>Not used only those</td>
</tr>
</tbody>
</table>

Directs assembler to list macro calls plus any lines generated in the expansion of the macro that generate data.
**NLSM (No Listing Of Macro Expansions)**

<table>
<thead>
<tr>
<th>Label</th>
<th>Operation Field</th>
<th>Variable Field</th>
</tr>
</thead>
<tbody>
<tr>
<td>Not used</td>
<td>NLSM</td>
<td>Not used</td>
</tr>
</tbody>
</table>

Inhibits listing of statements generated by the assembler during macro expansion. Only the macro call is listed.
BACK, BACK TO (Loop Back - Macros Only)

<table>
<thead>
<tr>
<th>Label</th>
<th>Operation Field</th>
<th>Variable Field</th>
</tr>
</thead>
<tbody>
<tr>
<td>Optional</td>
<td>BACK</td>
<td>Statement label</td>
</tr>
<tr>
<td></td>
<td>or BACK TO</td>
<td></td>
</tr>
</tbody>
</table>

This statement directs the assembler to repeat source statements that have already been assembled, beginning with the statement specified in the variable field. Such backward references are permitted only within a macro prototype. (Both the BACK, BACK TO and the specified statement label must lie between the same MAC statement and its corresponding ENDM).

Examples:

BACK TO AB16

BACK AB16

IF (X .NE. 0) BACK TO START+3

SAY (List Message to Operator)

<table>
<thead>
<tr>
<th>Label</th>
<th>Operation Field</th>
<th>Variable Field</th>
</tr>
</thead>
<tbody>
<tr>
<td>Optional</td>
<td>SAY</td>
<td>Any ASCII text string</td>
</tr>
</tbody>
</table>

The assembler responds to a SAY statement by printing the content of the variable field, starting at column 1 of the listing. Usually, the SAY statement is used within a macro to generate error comments or other messages to the operator. Macro argument references (enclosed by angle brackets) are replaced by their equivalent character string before output.

SAY statements generate output regardless of the setting of the listing options, as long as a listing device is assigned.
Macro Definition:

(0351) OLD MAC USING, AND
(0352) NLST
(0353) SAY
(0354) SAY *****************************************************
(0355) SAY OLD MACDONALD HAD A FARM, E-I-E-I-O.
(0356) SAY AND ON THIS FARM HE HAD SOME <2>, E-I-E-I-O.
(0357) SAY WITH A <2> <2> HERE AND A <2> <2> THERE,
(0358) SAY HERE A <2>, THERE A <2>,
(0359) SAY EVERYWHERE A <2> <2>,
(0360) SAY OLD MACDONALD HAD A FARM, E-I-E-I-O.
(0361) SAY *****************************************************
(0362) SAY
(0363) LIST
(0364) ENDM

Macro Calls:

(0365) OLD MAC, USING CHICKS AND CHEEP

***************************************************************
OLD MACDONALD HAD A FARM, E-I-E-I-O.
AND ON THIS FARM HE HAD SOME CHICKS, E-I-E-I-O.
WITH A CHEEP CHEEP HERE AND A CHEEP CHEEP THERE,
HERE A CHEEP, THERE A CHEEP,
EVERYWHERE A CHEEP CHEEP,
OLD MACDONALD HAD A FARM, E-I-E-I-O.
***************************************************************

(0366) OLD MAC, USING DUCKS AND QUACK

***************************************************************
OLD MACDONALD HAD A FARM, E-I-E-I-O.
AND ON THIS FARM HE HAD SOME DUCKS, E-I-E-I-O.
WITH A QUACK QUACK HERE AND A QUACK QUACK THERE,
HERE A QUACK, THERE A QUACK,
EVERYWHERE A QUACK QUACK,
OLD MACDONALD HAD A FARM, E-I-E-I-O.
***************************************************************

(0369) OLD MAC, USING NEWLYWEDS AND [BEEP]

***************************************************************
OLD MACDONALD HAD A FARM, E-I-E-I-O.
AND ON THIS FARM HE HAD SOME NEWLYWEDS, E-I-E-I-O.
WITH A [BEEP] [BEEP] HERE AND A [BEEP] [BEEP] THERE,
HERE A [BEEP], THERE A [BEEP],
EVERYWHERE A [BEEP] CHEEP,
OLD MACDONALD HAD A FARM, E-I-E-I-O.
***************************************************************
MACRO EXAMPLES

The following macro example makes use of local symbols, assembly attributes, and looping. The number of arguments processed by this macro is variable.

Definition:

```
ADJUST MAC BY, AND
SET $N = 1

@L1 IF ($N .GT. #2) GO TO @L2
LDA ($N) ARG. #1, 3, 5, ETC.
MPY ($N+1) ARG. #2, 4, 6, ETC.
STA ($N)
SET $N = $N + 2
BACK TO +L1

@L2 EBDN
```

Macro Calls: ADJUST A3 BY 16, A4 BY 20 AND A5 BY 3
ADJUST METER1 BY 100
ADJUST X1,2 X2,2 X3,2 X4,50 X5,50 X6,2

Definition:

```
MOVE MAC WORD, WORDS, FROM, TO
IF (<1> .EQ. 1) LDA <2>: STA <3>: GO TO $X
LDX = <1>
LDA <2> -1,1
STA <3> -1,1
DXS 1,1
JMP *-5
$X ENDM
```

5-15
Macro Calls:  MOVE  1, ALPHA, BETA
MOVE  20, LIST, TABLE
MOVE  1 WORD FROM ALPHA TO BETA
MOVE  20 WORDS FROM LIST TO TABLE
MOVE  3, FROM X31 TO Z21

The following macro does not generate any coding, just an answer. It demonstrates how macros can be used to construct interpreters as well as compilers.

Definition:

        FACTORIAL MAC OF
        FACTA <1>, 1
        ENDM

        *

        FACTA MAC
        IF (<1> .E0. 0) DATA <2>
        SET A2 = <1> * <2>, A1 = <1> - 1
        IF (<2> .NE. 0) FACTA A1,A2
        ENDM

Macro Calls:  FACTORIAL OF 5
              FACTORIAL 7
The following example shows the type of "language" that can be provided for business applications when a suitable set of macros are prepared in advance. (The macro definitions are not shown.)

*--------FILE AND FIELD DEFINITION.

INPUTREC FILE CONTAINS 80 WORDS FROM UNIT 5
OUTPUTREC FILE CONTAINS 80 WORDS ON UNIT 6

AMOUNT FIELD OF INPUTREC FROM COLUMNS 25 TO 20, 0 DECIMAL POSITIONS

CODE FIELD OF INPUTREC FROM COLUMNS 25 TO 30

NAME FIELD OF INPUTREC FROM COLUMNS 65 TO 75

NAMEOUT F IRLF OF OUTPUTREC FROM COLUMNS 1 TO 10

CODEOUT FIELD OF OUTPUTREC FROM COLUMNS 11 TO 16

AMOUNTOUT FIELD OF OUTPUTREC FROM COLUMNS 17 TO 23, 2 DECIMAL POSITIONS

*

*--------START EDITING PHASE OF PROGRAM.

START READ INPUTREC, IF END OF FILE, GO TO EOF

MOVE AMOUNT TO AMOUNTOUT

MOVE CODE TO CODEOUT

MOVE NAME TO NAMEOUT

WRITE OUTPUTREC

TO TO START

*

*--------FILE PROCESSING COMPLETE.

EOF END
SECTION 6
SOURCE FILE MERGING COMMANDS

The assembler includes a function called RDALN (read alternate lines) that has the ability to merge lines from two or more source files during assembly. File merging is controlled by special command statements in the primary source file that begin with a dollar sign:

$INSERT
$UPDATE
$COPY
$DONE

The primary source file is the file (or device) specified in the usual manner at the start of assembly. Secondary files stored on disk or mounted on a specific input-output device are selected by a filename or device code specified in the variable field of a $INSERT or $UPDATE command.

Only disk-resident files can be identified by filename. When non-disk devices are used, a device code is used instead of a filename:

(A) ASR       (P) PTR       (C) CARDS
(M) MAG TAPE   (K) Cassette

The parentheses must be included in device codes. Only the first character is needed to identify the device; other characters may be included for documentation (as in ASR or PAPER TAPE READER) but the extra characters are ignored.

File merging commands are allowed in the primary source file only, not in secondary files.

$INSERT

Label  Operation Field  Variable Field
Optional $INSERT        Filename or device code

When the assembler reaches an $INSERT command in the primary source file, it opens the specified file (or starts the device) and starts reading statements from the secondary file. The
secondary file is read in entirety, up to but not including the end-of-file mark. The assembler then returns to the primary file and resumes at the line following the $INSERT command. Nesting of $INSERT commands is not allowed.

$UPDATE

<table>
<thead>
<tr>
<th>Label</th>
<th>Operation Field</th>
<th>Variable Field</th>
</tr>
</thead>
<tbody>
<tr>
<td>Optional</td>
<td>$UPDATE</td>
<td>Filename or device code</td>
</tr>
</tbody>
</table>

When the assembler reaches an $UPDATE command in the primary source file, it opens the named file (or starts the device) but continues reading from the primary source file until a $COPY command is found in the primary file.

If the primary file contains more than one $UPDATE command, the one most recently processed determines the secondary file to be accessed by $COPY commands.

$COPY

<table>
<thead>
<tr>
<th>Label</th>
<th>Operation Field</th>
<th>Variable Field</th>
</tr>
</thead>
<tbody>
<tr>
<td>Optional</td>
<td>$COPY</td>
<td>m,n</td>
</tr>
</tbody>
</table>

When a $COPY command is found in a primary source file, lines m through n of the secondary file specified by the most recent $UPDATE command are input to the assembler. After line n has been processed, the assembler returns to the primary file and processes the record following the $COPY command.

$COPY commands may also be entered in the following forms:

- $COPY m       Copy line m of secondary file only
- $COPY ,n      Copy from current position in secondary file up to and including line n
<table>
<thead>
<tr>
<th>Label</th>
<th>Operation Field</th>
<th>Variable Field</th>
</tr>
</thead>
<tbody>
<tr>
<td>Optional</td>
<td>$DONE$</td>
<td>Not used</td>
</tr>
</tbody>
</table>

Upon reaching a $DONE$ statement, the assembler closes all open files and shuts down all devices used for secondary input.
APPENDICES

A  PRIME 200 Instructions (Op Code Order)  A-1
B  PRIME 200 Instructions (Class Order)   B-1
C  PRIME 200 Instructions (Mnemonic Order) C-1
D  I/O Device Codes                        D-1
E  ASCII Character Codes                   E-1
F  Object File Formats                     F-1
G  Assembler Error Messages                G-1
APPENDIX A
PRIME 200 INSTRUCTIONS
(OP CODE ORDER)

G 000000 HLT  HALT
G 000001 NOP  NO OPERATION
G 000005 SG1  ENTER SINGLE PRECISION MODE
G 000007 DBL  ENTER DOUBLE PRECISION MODE
G 000011 E16S(DXA) ENTER 16K SECTOR ADDRESSING MODE
G 000013 E32S(EXA) ENTER 32K SECTOR ADDRESSING MODE
G 000021 RMC(RMP) RESET MACHINE CHECK
G 000041 SCA  TRANSFER SHIFT COUNTER TO A
G 000043 INK  TRANSFER (INPUT) STATUS KEYS TO A
G 000101 NRM  NORMALIZE
G 000111 CEA  COMPUTE EFFECTIVE ADDRESS
G 000201 IAB  INTERCHANGE A AND B
G 000205 PIM  POSITION FOR INTEGER MULTIPLY
G 000211 PID  POSITION FOR INTEGER DIVIDE
G 000311** VIRY  VERIFY
G 000401 ENB  ENABLE INTERRUPT
G 000405 OTK  TRANSFER (OUTPUT) A TO STATUS KEYS
G 000411 CAI  CLEAR ACTIVE INTERRUPT
G 000415 ESIM  ENTER STANDARD INTERRUPT MODE
G 000417 EVIM  ENTER VECTORED INTERRUPT MODE
G 000501 LMCM  LEAVE MACHINE CHECK MODE
G 000503 ENCMM ENTER MACHINE CHECK MODE
G 000505 SVC  SUPERVISOR CALL
G 000511 ISI  INPUT SERIAL INTERFACE TO A
G 000515 OSI  OUTPUT SERIAL INTERFACE FROM A
G 001001 INH  INHIBIT INTERRUPT
G 001013 E32R  ENTER 32K RELATIVE ADDRESSING MODE
MR 01 JMP  UNCONDITIONAL JUMP
MR 02 * DLD  DOUBLE PRECISION LOAD
MR 02  LDA  LOAD A
MR 03  ANA  AND TO A
MR 04 * DST  DOUBLE PRECISION STORE
MR 04  STA  STORE A
SH 0400NN LRL  LONG RIGHT LOGICAL
SH 0401NN LRS  LONG RIGHT SHIFT
SH 0402NN LRR  LONG RIGHT ROTATE
SH 0404NN ARL(LGR) A RIGHT LOGICAL
SH 0405NN ARS  A RIGHT SHIFT
SH 0406NN ARR  A RIGHT ROTATE
SH 0410NN LLL  LONG LEFT LOGICAL
SH 0411NN LLS  LONG LEFT SHIFT
SH 0412NN LLR  LONG LEFT ROTATE
SH 0414NN ALL (LGL) A LEFT LOGICAL
SH 0415NN ALS A LEFT SHIFT
SH 0416NN ALR A LEFT ROTATE
MR 05 ERA EXCLUSIVE OR TO A
MR 06 * DAD DOUBLE PRECISION ADD
MR 06 ADD ADD MEMORY TO A
MR 07 * DSB DOUBLE PRECISION SUBTRACT
MR 07 SUB SUBTRACT MEMORY FROM A
MR 10 JST JUMP TO EA + 1 AND STORE P IN EA
G 100000 SKP UNCONDITIONAL SKIP
G 100001 SRC SKIP ON C-BIT RESET
G 100036 SSR SKIP ON NONE OF SENSE SWITCHES 1-4 SET
G 100040 SZE (SEQ) SKIP ON A ZERO
G 100100 SLZ SKIP ON A BIT 16 ZERO
G 100200 SMCR(SPN) SKIP ON MACHINE CHECK RESET
G 100220 SGT SKIP ON A GREATER THAN ZERO
BR 100240+N SRN SKIP ON SENSE SWITCH N RESET
BR 100260+N SARN SKIP ON A BIT N RESET
G 100400 SPL (SGE) SKIP ON A PLUS
G 101001 SSC SKIP ON C-BIT SET
G 101036 SSS SKIP ON ANY OF SENSE SWITCHES 1-4 SET
G 101040 SNZ (SNE) SKIP ON A NOT ZERO
G 101100 SLN SKIP ON A BIT 16 ONE
G 101200 SMCS(SPS) SKIP ON MACHINE CHECK SET
G 101220 SLE SKIP ON A LESS THAN OR EQUAL TO ZERO
BR 101240+N SSN SKIP ON SENSE SWITCH N SET
BR 101260+N SASN SKIP ON A BIT N SET
G 101400 SMI (SLT) SKIP ON A MINUS
MR 11 CAS COMPARE A WITH MEMORY
MR 12 IRS INCREMENT, REPLACE MEMORY AND SKIP
MR 13 IMA INTERCHANGE MEMORY AND A
10 14 OCP OUTPUT CONTROL PULSE
G 140010 CRL CLEAR LONG (A AND B)
G 140014 CRB CLEAR B
G 140024 CHS CHANGE SIGN OF A
G 140040 CRA CLEAR A
G 140100 SSP SET SIGN OF A PLUS
G 140104 XCA TRANSFER A TO B AND CLEAR A
G 140110 SOA (SIA) SUBTRACT ONE FROM A
G 140114 IRX INCREMENT, REPLACE INDEX AND SKIP
G 140200 RCB RESET C-BIT
G 140204 XCB TRANSFER B TO A AND CLEAR B
G 140210 DRX DECREMENT REPLACE INDEX AND SKIP
G 140214 CAZ COMPARE A WITH ZERO
G 140304 A2A ADD TWO TO A
G 140310 S2A SUBTRACT TWO FROM A
G 140320 CSA COPY SIGN TO C-BIT, SET SIGN OF A PLUS
G 140401 CMA COMPLEMENT A
G 140407 TCA TWO’S COMPLEMENT A
G 140410 LLT CONVERT A<0 TO TRUE
G 140411 LLE CONVERT A<=0 TO TRUE
G 140412 LNE CONVERT A<0 TO TRUE

A-2
G  140413  LEQ  CONVERT A=0 TO TRUE
G  140414  LGE  CONVERT A>=0 TO TRUE
G  140415  LGT  CONVERT A>0 TO TRUE
G  140500  SSM  SET SIGN OF A MINUS
G  140600  SCB  SET C-BIT
G  141044  CAR  CLEAR RIGHT BYTE OF A
G  141050  CAL  CLEAR LEFT BYTE OF A
G  141140  ICL  INTERCHANGE BYTES OF A AND CLEAR LEFT BYTE
G  141206  AOA (A1A) ADD ONE TO A
G  141216  ACA  ADD C-BIT TO A
G  141240  ICR  INTERCHANGE BYTES OF A AND CLEAR RIGHT BYTE
G  141340  ICA  INTERCHANGE BYTES OF A
MR  15   LDX  LOAD INDEX (ASSEMBLER SETS INDEX BIT)
MR  15   STX  STORE INDEX (ASSEMBLER CLEAR INDEX BIT)
MR  16   MPY  MULTIPLY
MR  17   DIV  DIVIDE
IO  34   SKS  SKIP IF SET
IO  54   INA  INPUT TO A
IO  74   OTA  OUTPUT FROM A
IO  74   SMK  SET INTERRUPT MASK
### APPENDIX B
### PRIME 200 INSTRUCTIONS

#### (CLASS ORDER)

<table>
<thead>
<tr>
<th>Code</th>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BR</td>
<td>100240+N SRN</td>
<td>SKIP ON SENSE SWITCH N RESET</td>
</tr>
<tr>
<td>BR</td>
<td>100260+N SARN</td>
<td>SKIP ON A BIT N RESET</td>
</tr>
<tr>
<td>BR</td>
<td>101240+N SSN</td>
<td>SKIP ON SENSE SWITCH N SET</td>
</tr>
<tr>
<td>BR</td>
<td>101260+N SASN</td>
<td>SKIP ON A BIT N SET</td>
</tr>
<tr>
<td>G</td>
<td>000000 HLT</td>
<td>HALT</td>
</tr>
<tr>
<td>G</td>
<td>000001 NOP</td>
<td>NO OPERATION</td>
</tr>
<tr>
<td>G</td>
<td>000005 SGL</td>
<td>ENTER SINGLE PRECISION MODE</td>
</tr>
<tr>
<td>G</td>
<td>000007 DPL</td>
<td>ENTER DOUBLE PRECISION MODE</td>
</tr>
<tr>
<td>G</td>
<td>000011 E16S(DXA)</td>
<td>ENTER 16K SECTOR ADDRESSING MODE</td>
</tr>
<tr>
<td>G</td>
<td>000013 E32S(EXA)</td>
<td>ENTER 32K SECTOR ADDRESSING MODE</td>
</tr>
<tr>
<td>G</td>
<td>000021 RMC&lt;REM&gt;</td>
<td>RESET MACHINE CHECK</td>
</tr>
<tr>
<td>G</td>
<td>000041 SCA</td>
<td>TRANSFER SHIFT COUNTER TO A</td>
</tr>
<tr>
<td>G</td>
<td>000043 INK</td>
<td>TRANSFER (INPUT) STATUS KEYS TO A</td>
</tr>
<tr>
<td>G</td>
<td>000101 NRM</td>
<td>NORMALIZE</td>
</tr>
<tr>
<td>G</td>
<td>000111 CEA</td>
<td>COMPUTE EFFECTIVE ADDRESS</td>
</tr>
<tr>
<td>G</td>
<td>000201 IAB</td>
<td>INTERCHANGE A AND B</td>
</tr>
<tr>
<td>G</td>
<td>000205 PIM</td>
<td>POSITION FOR INTEGER MULTIPLY</td>
</tr>
<tr>
<td>G</td>
<td>000211 PID</td>
<td>POSITION FOR INTEGER DIVIDE</td>
</tr>
<tr>
<td>G</td>
<td>000311** VIRY</td>
<td>VERIFY</td>
</tr>
<tr>
<td>G</td>
<td>000401 ENB</td>
<td>ENABLE INTERRUPT</td>
</tr>
<tr>
<td>G</td>
<td>000405 OTK</td>
<td>TRANSFER (OUTPUT) A TO STATUS KEYS</td>
</tr>
<tr>
<td>G</td>
<td>000411 CAI</td>
<td>CLEAR ACTIVE INTERRUPT</td>
</tr>
<tr>
<td>G</td>
<td>000415 ESIM</td>
<td>ENTER STANDARD INTERRUPT MODE</td>
</tr>
<tr>
<td>G</td>
<td>000417 EVIM</td>
<td>ENTER VECTORED INTERRUPT MODE</td>
</tr>
<tr>
<td>G</td>
<td>000501 LMCM</td>
<td>LEAVE MACHINE CHECK MODE</td>
</tr>
<tr>
<td>G</td>
<td>000503 EMCM</td>
<td>ENTER MACHINE CHECK MODE</td>
</tr>
<tr>
<td>G</td>
<td>000505 SVC</td>
<td>SUPERVISOR CALL</td>
</tr>
<tr>
<td>G</td>
<td>000511 ISI</td>
<td>INPUT SERIAL INTERFACE TO A</td>
</tr>
<tr>
<td>G</td>
<td>000515 OSI</td>
<td>OUTPUT SERIAL INTERFACE FROM A</td>
</tr>
<tr>
<td>G</td>
<td>001001 INH</td>
<td>INHIBIT INTERRUPT</td>
</tr>
<tr>
<td>G</td>
<td>001013 E32R</td>
<td>ENTER 32K RELATIVE ADDRESSING MODE</td>
</tr>
<tr>
<td>G</td>
<td>100000 SKP</td>
<td>UNCONDITIONAL SKIP</td>
</tr>
<tr>
<td>G</td>
<td>100001 SRC</td>
<td>SKIP ON C-BIT RESET</td>
</tr>
<tr>
<td>G</td>
<td>100036 SSR</td>
<td>SKIP ON NONE OF SENSE SWITCHES 1-4 SET</td>
</tr>
<tr>
<td>G</td>
<td>100040 SZE&lt;SEO&gt;</td>
<td>SKIP ON A ZERO</td>
</tr>
<tr>
<td>G</td>
<td>100100 SLZ</td>
<td>SKIP ON A BIT 16 ZERO</td>
</tr>
<tr>
<td>G</td>
<td>100200 SMCR&lt;SPN</td>
<td>SKIP ON MACHINE CHECK RESET</td>
</tr>
<tr>
<td>G</td>
<td>100220 SGT</td>
<td>SKIP ON A GREATER THAN ZERO</td>
</tr>
<tr>
<td>G</td>
<td>100400 SPL&lt;SGE</td>
<td>SKIP ON A PLUS</td>
</tr>
<tr>
<td>G</td>
<td>101001 SSC</td>
<td>SKIP ON C-BIT SET</td>
</tr>
<tr>
<td>G</td>
<td>101036 SSS</td>
<td>SKIP ON ANY OF SENSE SWITCHES 1-4 SET</td>
</tr>
<tr>
<td>G</td>
<td>101040 SNZ&lt;SNE&gt;</td>
<td>SKIP ON A NOT ZERO</td>
</tr>
<tr>
<td>G</td>
<td>101100 SLN</td>
<td>SKIP ON A BIT 16 ONE</td>
</tr>
<tr>
<td>G</td>
<td>101200 SMCS&lt;SPS&gt;</td>
<td>SKIP ON MACHINE CHECK SET</td>
</tr>
</tbody>
</table>

B-1
10220  SLE  SKIP ON A LESS THAN OR EQUAL TO ZERO
10400  SMI  (SLT)  SKIP ON A MINUS
14000  CLR  CLEAR LONG (A AND B)
14004  CIR  CLEAR B
14002  CHS  CHANGE SIGN OF A
14004  CRA  CLEAR A
14010  SSM  SET SIGN OF A PLUS
14010  XCA  TRANSFER A TO B AND CLEAR A
14010  SDA  (S1A)  SUBTRACT ONE FROM A
14010  IRX  INCREMENT, REPLACE INDEX AND SKIP
14020  RCX  CLEAR C-BIT
14020  XCB  TRANSFER B TO A AND CLEAR B
14020  DRC  DECIMMENT REPLACE INDEX AND SKIP
14020  CAZ  COMPARE A WITH ZERO
14030  A2A  ADD TWO TO A
14030  S2A  SUBTRACT TWO FROM A
14030  CSA  COPY SIGN TO C-BIT, SET SIGN OF A PLUS
14040  CKP  COMPLEMENT A
14040  TCA  TWO'S COMPLEMENT A
14040  LCT  CONVERT A<0 TO TRUE
14040  LEF  CONVERT A<0 TO TRUE
14040  LNE  CONVERT 'A=0' TO TRUE
14040  LOE  CONVERT A=0 TO TRUE
14040  LGE  CONVERT A>0 TO TRUE
14040  LGT  CONVERT A>0 TO TRUE
14050  SSM  SET SIGN OF A MINUS
14060  SCB  SET C-BIT
14100  CAR  CLEAR RIGHT BYTE OF A
14100  CAL  CLEAR LEFT BYTE OF A
14140  TCL  INTERCHANGE BYTES OF A AND CLEAR LEFT BYTE
14200  AOA  (A1A)  ADD ONE TO A
14216  ACA  ADD C-BIT TO A
14240  ICR  INTERCHANGE BYTES OF A AND CLEAR RIGHT BYTE
14340  ICA  INTERCHANGE BYTES OF A
10  14  DCP  OUTPUT CONTROL PULSE
10  34  SKS  SKIP IF SET
10  54  INA  INPUT TO A
10  74  OTA  OUTPUT FROM A
10  74  SMK  SET INTERRUPT MASK
MR  01  JMP  UNCONDITIONAL JUMP
MR  02  *  DLD  DOUBLE PRECISION LOAD
MR  02  LDA  LOAD A
MR  03  *  AND  AND TO A
MR  04  *  DST  DOUBLE PRECISION STORE
MR  04  STA  STORE A
MR  05  ERA  EXCLUSIVE OR TO A
MR  06  *  DAD  DOUBLE PRECISION ADD
MR  06  *  ADD  ADD MEMORY TO A
MR  07  *  DSB  DOUBLE PRECISION SUBTRACT
MR  07  SUB  SUBTRACT MEMORY FROM A
MR  10  JST  JUMP TO EA + 1 AND STORE P IN EA
MR  11  CAS  COMPARE A WITH MEMORY
MR  12  IRS  INCREMENT, REPLACE MEMORY AND SKIP

B-2
<table>
<thead>
<tr>
<th>MR</th>
<th>13</th>
<th>IMA</th>
<th>INTERCHANGE MEMORY AND A</th>
</tr>
</thead>
<tbody>
<tr>
<td>MR</td>
<td>15</td>
<td>LDX</td>
<td>LOAD INDEX (ASSEMBLER SETS INDEX BIT)</td>
</tr>
<tr>
<td>MR</td>
<td>15</td>
<td>STX</td>
<td>STORE INDEX (ASSEMBLER CLEARS INDEX BIT)</td>
</tr>
<tr>
<td>MR</td>
<td>16</td>
<td>MPY</td>
<td>MULTIPLY</td>
</tr>
<tr>
<td>MR</td>
<td>17</td>
<td>DIV</td>
<td>DIVIDE</td>
</tr>
<tr>
<td>SH</td>
<td>0400NN</td>
<td>LRL</td>
<td>LONG RIGHT LOGICAL</td>
</tr>
<tr>
<td>SH</td>
<td>0401NN</td>
<td>LRS</td>
<td>LONG RIGHT SHIFT</td>
</tr>
<tr>
<td>SH</td>
<td>0402NN</td>
<td>LRR</td>
<td>LONG RIGHT ROTATE</td>
</tr>
<tr>
<td>SH</td>
<td>0404NN</td>
<td>ARL (LGR)</td>
<td>A RIGHT LOGICAL</td>
</tr>
<tr>
<td>SH</td>
<td>0405NN</td>
<td>ARS</td>
<td>A RIGHT SHIFT</td>
</tr>
<tr>
<td>SH</td>
<td>0406NN</td>
<td>ARR</td>
<td>A RIGHT ROTATE</td>
</tr>
<tr>
<td>SH</td>
<td>0410NN</td>
<td>LLL</td>
<td>LONG LEFT LOGICAL</td>
</tr>
<tr>
<td>SH</td>
<td>0411NN</td>
<td>LLS</td>
<td>LONG LEFT SHIFT</td>
</tr>
<tr>
<td>SH</td>
<td>0412NN</td>
<td>LLR</td>
<td>LONG LEFT ROTATE</td>
</tr>
<tr>
<td>SH</td>
<td>0414NN</td>
<td>ALL (LGL)</td>
<td>A LEFT LOGICAL</td>
</tr>
<tr>
<td>SH</td>
<td>0415NN</td>
<td>ALS</td>
<td>A LEFT SHIFT</td>
</tr>
<tr>
<td>SH</td>
<td>0416NN</td>
<td>ALR</td>
<td>A LEFT ROTATE</td>
</tr>
</tbody>
</table>
APPENDIX C

PRIME 200 INSTRUCTIONS

(MNEMONIC ORDER)

G 140304  A2A ADD TWO TO A
G 141216  ACA ADD C-BIT TO A
MR 06  ADD ADD MEMORY TO A
SH 0414NN  ALL (LGL) A LEFT LOGICAL
SH 0416NN  ALR A LEFT ROTATE
SH 0415NN  ALS A LEFT SHIFT
MR 03  ANA AND TO A
G 141206  AOA (A+1A) ADD ONE TO A
SH 0404NN  ARL (LGR) A RIGHT LOGICAL
SH 0406NN  ARR A RIGHT ROTATE
SH 0405NN  ARS A RIGHT SHIFT
G 000411  CAT CLEAR ACTIVE INTERRUPT
G 141050  CAL CLEAR LEFT BYTE OF A
G 141044  CAR CLEAR RIGHT BYTE OF A
MR 11  CAS COMPARE A WITH MEMORY
G 140214  CA2 COMPARE A WITH ZERO
G 000111  CEA COMPUTE EFFECTIVE ADDRESS
G 140024  CHS CHANGE SIGN OF A
G 140401  CMN COMPLEMENT A
G 140040  CRA CLEAR A
G 140014  CRB CLEAR B
G 140010  CRL CLEAR LONG (A AND B)
G 140320  CSA COPY SIGN TO C-BIT, SET SIGN OF A PLUS
MR 06  DAD DOUBLE PRECISION ADD
G 000007  DBL ENTER DOUBLE PRECISION MODE
MR 17  DIV DIVIDE
MR 02  DLD DOUBLE PRECISION LOAD
G 140210  DRX DECREMENT REPLACE INDEX AND SKIP
MR 07  DSU DOUBLE PRECISION SUBTRACT
MR 04  DST DOUBLE PRECISION STORE
G 000011  E16S(DXA) ENTER 16K SECTOR ADDRESSING MODE
G 001013  E2R ENTER 32K RELATIVE ADDRESSING MODE
G 000013  E2S(EXA) ENTER 32K SECTOR ADDRESSING MODE
G 000503  EMCM ENTER MACHINE CHECK MODE
G 000401  ENB ENABLE INTERRUPT
MR 05  ERA EXCLUSIVE OR TO A
G 000415  ESTM ENTER STANDARD INTERRUPT MODE
G 000417  EVIM ENTER VECTORED INTERRUPT MODE
G 000000  HLT HALT
G 000201  IAB INTERCHANGE A AND B
G 141340  ICA INTERCHANGE BYTES OF A
G 141140  ICL INTERCHANGE BYTES OF A AND CLEAR LEFT BYTE
C 141240  ICR INTERCHANGE BYTES OF A AND CLEAR RIGHT BYT
<table>
<thead>
<tr>
<th>MR</th>
<th>13</th>
<th>IMA</th>
<th>INTERCHANGE MEMORY AND A</th>
</tr>
</thead>
<tbody>
<tr>
<td>IO</td>
<td>54</td>
<td>INA</td>
<td>INPUT TO A</td>
</tr>
<tr>
<td>G</td>
<td>001001</td>
<td>INH</td>
<td>INHIBIT INTERRUPT</td>
</tr>
<tr>
<td>G</td>
<td>000043</td>
<td>INK</td>
<td>TRANSFER (INPUT) STATUS KEYS TO A</td>
</tr>
<tr>
<td>MR</td>
<td>12</td>
<td>IRS</td>
<td>INCREMENT, REPLACE MEMORY AND SKIP</td>
</tr>
<tr>
<td>G</td>
<td>140114</td>
<td>IRX</td>
<td>INCREMENT, REPLACE INDEX AND SKIP</td>
</tr>
<tr>
<td>G</td>
<td>000511</td>
<td>ISI</td>
<td>INPUT SERIAL INTERFACE TO A</td>
</tr>
<tr>
<td>MR</td>
<td>01</td>
<td>JMP</td>
<td>UNCONDITIONAL JUMP</td>
</tr>
<tr>
<td>MR</td>
<td>10</td>
<td>JST</td>
<td>JUMP TO EA + 1 AND STORE P IN EA</td>
</tr>
<tr>
<td>MR</td>
<td>02</td>
<td>LDA</td>
<td>LOAD A</td>
</tr>
<tr>
<td>MR</td>
<td>15</td>
<td>LDX</td>
<td>LOAD INDEX (ASSEMBLER SETS INDEX BIT)</td>
</tr>
<tr>
<td>G</td>
<td>140413</td>
<td>LEQ</td>
<td>CONVERT A=0 TO TRUE</td>
</tr>
<tr>
<td>G</td>
<td>140414</td>
<td>LGE</td>
<td>CONVERT A≥0 TO TRUE</td>
</tr>
<tr>
<td>G</td>
<td>140415</td>
<td>LGT</td>
<td>CONVERT A&gt;0 TO TRUE</td>
</tr>
<tr>
<td>G</td>
<td>140411</td>
<td>LLE</td>
<td>CONVERT A&lt;0 TO TRUE</td>
</tr>
<tr>
<td>SH</td>
<td>0410NN</td>
<td>LLL</td>
<td>LONG LEFT LOGICAL</td>
</tr>
<tr>
<td>SH</td>
<td>0412NN</td>
<td>LLR</td>
<td>LONG LEFT ROTATE</td>
</tr>
<tr>
<td>SH</td>
<td>0411NN</td>
<td>LLS</td>
<td>LONG LEFT SHIFT</td>
</tr>
<tr>
<td>G</td>
<td>140410</td>
<td>LLT</td>
<td>CONVERT A&lt;0 TO TRUE</td>
</tr>
<tr>
<td>G</td>
<td>000501</td>
<td>LMCM</td>
<td>LEAVE MACHINE CHECK MODE</td>
</tr>
<tr>
<td>G</td>
<td>140412</td>
<td>LNE</td>
<td>CONVERT 'A(=0) TO TRUE</td>
</tr>
<tr>
<td>SH</td>
<td>0406NN</td>
<td>LRL</td>
<td>LONG RIGHT LOGICAL</td>
</tr>
<tr>
<td>SH</td>
<td>0402NN</td>
<td>LRR</td>
<td>LONG RIGHT ROTATE</td>
</tr>
<tr>
<td>SH</td>
<td>0401NN</td>
<td>LRS</td>
<td>LONG RIGHT SHIFT</td>
</tr>
<tr>
<td>MR</td>
<td>16</td>
<td>MPY</td>
<td>MULTIPLY</td>
</tr>
<tr>
<td>G</td>
<td>000001</td>
<td>NOP</td>
<td>NO OPERATION</td>
</tr>
<tr>
<td>G</td>
<td>000101</td>
<td>NRM</td>
<td>NORMALIZE</td>
</tr>
<tr>
<td>IO</td>
<td>14</td>
<td>OCP</td>
<td>OUTPUT CONTROL PULSE</td>
</tr>
<tr>
<td>G</td>
<td>000515</td>
<td>OSI</td>
<td>OUTPUT SERIAL INTERFACE FROM A</td>
</tr>
<tr>
<td>IO</td>
<td>74</td>
<td>OTA</td>
<td>OUTPUT FROM A</td>
</tr>
<tr>
<td>G</td>
<td>000405</td>
<td>OTK</td>
<td>TRANSFER (OUTPUT) A TO STATUS KEYS</td>
</tr>
<tr>
<td>G</td>
<td>000211</td>
<td>PID</td>
<td>POSITION FOR INTEGER DIVIDE</td>
</tr>
<tr>
<td>G</td>
<td>000205</td>
<td>PIN</td>
<td>POSITION FOR INTEGER MULTIPLY</td>
</tr>
<tr>
<td>G</td>
<td>140200</td>
<td>RCB</td>
<td>RESET C-BIT</td>
</tr>
<tr>
<td>G</td>
<td>000021</td>
<td>RMC</td>
<td>RESET MACHINE CHECK</td>
</tr>
<tr>
<td>G</td>
<td>140310</td>
<td>S2A</td>
<td>SUBTRACT TWO FROM A</td>
</tr>
<tr>
<td>BR</td>
<td>100260+N SARN</td>
<td>SKIP ON A BIT N RESET</td>
<td></td>
</tr>
<tr>
<td>BR</td>
<td>101260+N SASN</td>
<td>SKIP ON A BIT N SET</td>
<td></td>
</tr>
<tr>
<td>G</td>
<td>000041</td>
<td>SCA</td>
<td>TRANSFER SHIFT COUNTER TO A</td>
</tr>
<tr>
<td>G</td>
<td>140600</td>
<td>SCB</td>
<td>SET C-BIT</td>
</tr>
<tr>
<td>G</td>
<td>000005</td>
<td>SGL</td>
<td>ENTER SINGLE PRECISION MODE</td>
</tr>
<tr>
<td>G</td>
<td>100220</td>
<td>SGT</td>
<td>SKIP ON A GREATER THAN ZERO</td>
</tr>
<tr>
<td>G</td>
<td>100000</td>
<td>SKP</td>
<td>UNCONDITIONAL SKIP</td>
</tr>
<tr>
<td>IO</td>
<td>34</td>
<td>SKS</td>
<td>SKIP IF SET</td>
</tr>
<tr>
<td>G</td>
<td>101220</td>
<td>SLE</td>
<td>SKIP ON A LESS THAN OR EQUAL TO ZERO</td>
</tr>
<tr>
<td>G</td>
<td>101110</td>
<td>SLN</td>
<td>SKIP ON A BIT 16 ONE</td>
</tr>
<tr>
<td>G</td>
<td>100100</td>
<td>SLZ</td>
<td>SKIP ON A BIT 16 ZERO</td>
</tr>
<tr>
<td>G</td>
<td>100200</td>
<td>SMCRSPN</td>
<td>SKIP ON MACHINE CHECK RESET</td>
</tr>
<tr>
<td>G</td>
<td>101200</td>
<td>SMCSSPS</td>
<td>SKIP ON MACHINE CHECK SET</td>
</tr>
<tr>
<td>G</td>
<td>101400</td>
<td>SMI SLT</td>
<td>SKIP ON A MINUS</td>
</tr>
<tr>
<td>IO</td>
<td>74</td>
<td>SMK</td>
<td>SET INTERRUPT MASK</td>
</tr>
<tr>
<td>G</td>
<td>101040</td>
<td>SNZ SNE</td>
<td>SKIP ON A NOT ZERO</td>
</tr>
<tr>
<td>G</td>
<td>140110</td>
<td>SOA S1A</td>
<td>SUBTRACT ONE FROM A</td>
</tr>
<tr>
<td>G</td>
<td>100400</td>
<td>SPL SGE</td>
<td>SKIP ON A PLUS</td>
</tr>
<tr>
<td>G</td>
<td>100001</td>
<td>SRC</td>
<td>SKIP ON C-BIT RESET</td>
</tr>
<tr>
<td>BR</td>
<td>100240+N SRN</td>
<td>SKIP ON SENSE SWITCH N RESET</td>
<td></td>
</tr>
<tr>
<td>Code</td>
<td>Description</td>
<td></td>
<td></td>
</tr>
<tr>
<td>------</td>
<td>-------------------------------------</td>
<td></td>
<td></td>
</tr>
<tr>
<td>G</td>
<td>101001  SSC  SKIP ON C-BIT SET</td>
<td></td>
<td></td>
</tr>
<tr>
<td>G</td>
<td>140500  SSM  SET SIGN OF A MINUS</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BR</td>
<td>101240+N SSN  SKIP ON SENSE SWITCH N SET</td>
<td></td>
<td></td>
</tr>
<tr>
<td>G</td>
<td>140100  SSP  SET SIGN OF A PLUS</td>
<td></td>
<td></td>
</tr>
<tr>
<td>G</td>
<td>100036  SSR  SKIP ON NONE OF SENSE SWITCHES 1-4 SET</td>
<td></td>
<td></td>
</tr>
<tr>
<td>G</td>
<td>101036  SSS  SKIP ON ANY OF SENSE SWITCHES 1-4 SET</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MR</td>
<td>04      STA  STORE A</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MR</td>
<td>15      STX  STORE INDEX (ASSEMBLER SETS INDEX BIT)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MR</td>
<td>07      SUB  SUBTRACT MEMORY FROM A</td>
<td></td>
<td></td>
</tr>
<tr>
<td>G</td>
<td>000505  SVC  SUPERVISOR CALL</td>
<td></td>
<td></td>
</tr>
<tr>
<td>G</td>
<td>100040  SZE (SEQ)  SKIP ON A ZERO</td>
<td></td>
<td></td>
</tr>
<tr>
<td>G</td>
<td>140407  TCA  TWO'S COMPLEMENT A</td>
<td></td>
<td></td>
</tr>
<tr>
<td>G</td>
<td>000311** VIRY  VERIFY</td>
<td></td>
<td></td>
</tr>
<tr>
<td>G</td>
<td>140104  XCA  TRANSFER A TO B AND CLEAR A</td>
<td></td>
<td></td>
</tr>
<tr>
<td>G</td>
<td>140204  XCB  TRANSFER B TO A AND CLEAR B</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
APPENDIX D
I/O DEVICE CODES
### APPENDIX E

**ASCII CHARACTER CODES**

<table>
<thead>
<tr>
<th>Character</th>
<th>Octal Code</th>
<th>Character</th>
<th>Octal Code</th>
<th>Character</th>
<th>Octal Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>260</td>
<td>X</td>
<td>330</td>
<td>SOM</td>
<td>201</td>
</tr>
<tr>
<td>1</td>
<td>261</td>
<td>Y</td>
<td>331</td>
<td>EOA</td>
<td>202</td>
</tr>
<tr>
<td>2</td>
<td>262</td>
<td>Z</td>
<td>332</td>
<td>EOM</td>
<td>203</td>
</tr>
<tr>
<td>3</td>
<td>263</td>
<td>(blank)</td>
<td>240</td>
<td>EOT</td>
<td>204</td>
</tr>
<tr>
<td>4</td>
<td>264</td>
<td></td>
<td>241</td>
<td>WRU</td>
<td>205</td>
</tr>
<tr>
<td>5</td>
<td>265</td>
<td>&quot;</td>
<td>242</td>
<td>RU</td>
<td>206</td>
</tr>
<tr>
<td>6</td>
<td>266</td>
<td>#</td>
<td>243</td>
<td>BEL</td>
<td>207</td>
</tr>
<tr>
<td>7</td>
<td>267</td>
<td>$</td>
<td>244</td>
<td>FE</td>
<td>210</td>
</tr>
<tr>
<td>8</td>
<td>270</td>
<td></td>
<td>245</td>
<td>H TAB</td>
<td>211</td>
</tr>
<tr>
<td>9</td>
<td>271</td>
<td>&amp;</td>
<td>246</td>
<td>LINE FEED</td>
<td>212</td>
</tr>
<tr>
<td>A</td>
<td>301</td>
<td>'</td>
<td>247</td>
<td>V TAB</td>
<td>213</td>
</tr>
<tr>
<td>B</td>
<td>302</td>
<td>(</td>
<td>250</td>
<td>FORM</td>
<td>214</td>
</tr>
<tr>
<td>C</td>
<td>303</td>
<td>)</td>
<td>251</td>
<td>RETURN</td>
<td>215</td>
</tr>
<tr>
<td>D</td>
<td>304</td>
<td>*</td>
<td>252</td>
<td>SO</td>
<td>216</td>
</tr>
<tr>
<td>E</td>
<td>305</td>
<td>+</td>
<td>253</td>
<td>SI</td>
<td>217</td>
</tr>
<tr>
<td>F</td>
<td>206</td>
<td>;</td>
<td>254</td>
<td>DCO</td>
<td>220</td>
</tr>
<tr>
<td>G</td>
<td>307</td>
<td>–</td>
<td>255</td>
<td>X-ON</td>
<td>221</td>
</tr>
<tr>
<td>H</td>
<td>310</td>
<td>.</td>
<td>256</td>
<td>TAPE AUX</td>
<td>222</td>
</tr>
<tr>
<td>I</td>
<td>311</td>
<td>/</td>
<td>257</td>
<td>ON</td>
<td>223</td>
</tr>
<tr>
<td>J</td>
<td>312</td>
<td>:</td>
<td>272</td>
<td>X-OFF</td>
<td>224</td>
</tr>
<tr>
<td>K</td>
<td>313</td>
<td>;</td>
<td>273</td>
<td>TAPE OFF</td>
<td>225</td>
</tr>
<tr>
<td>L</td>
<td>314</td>
<td></td>
<td>274</td>
<td>AUX</td>
<td>226</td>
</tr>
<tr>
<td>M</td>
<td>315</td>
<td>=</td>
<td>275</td>
<td>ERROR</td>
<td>227</td>
</tr>
<tr>
<td>N</td>
<td>316</td>
<td></td>
<td>276</td>
<td>SYNC</td>
<td>228</td>
</tr>
<tr>
<td>O</td>
<td>317</td>
<td>@</td>
<td>277</td>
<td>LEM</td>
<td>229</td>
</tr>
<tr>
<td>P</td>
<td>320</td>
<td>#</td>
<td>300</td>
<td>SO</td>
<td>230</td>
</tr>
<tr>
<td>Q</td>
<td>321</td>
<td></td>
<td>333</td>
<td>S1</td>
<td>231</td>
</tr>
<tr>
<td>R</td>
<td>322</td>
<td></td>
<td>334</td>
<td>S2</td>
<td>232</td>
</tr>
<tr>
<td>S</td>
<td>323</td>
<td></td>
<td>335</td>
<td>S3</td>
<td>233</td>
</tr>
<tr>
<td>T</td>
<td>324</td>
<td></td>
<td>336</td>
<td>S4</td>
<td>234</td>
</tr>
<tr>
<td>U</td>
<td>325</td>
<td></td>
<td>337</td>
<td>S5</td>
<td>235</td>
</tr>
<tr>
<td>V</td>
<td>326</td>
<td>RUBOUT</td>
<td>377</td>
<td>S6</td>
<td>236</td>
</tr>
<tr>
<td>W</td>
<td>327</td>
<td>NUL</td>
<td>200</td>
<td>S7</td>
<td>237</td>
</tr>
</tbody>
</table>
APPENDIX F

OBJECT FILE FORMATS
### APPENDIX G

**ASSEMBLER ERROR MESSAGES**

<table>
<thead>
<tr>
<th>Code</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>F</td>
<td>Macro argument number not found, unrecognized operand type, or FAIL pseudo-op executed.</td>
</tr>
<tr>
<td>G</td>
<td>Improper GO TO reference; or END or ENDM within a skip area.</td>
</tr>
<tr>
<td>I</td>
<td>Improper indirect flag.</td>
</tr>
<tr>
<td>L</td>
<td>Improper label, or external label in a literal, or missing label.</td>
</tr>
<tr>
<td>M</td>
<td>Multiply defined.</td>
</tr>
<tr>
<td>N</td>
<td>END within a Macro definition or an IF area.</td>
</tr>
<tr>
<td>O</td>
<td>Unrecognized Operator.</td>
</tr>
<tr>
<td>P</td>
<td>Mismatched parentheses.</td>
</tr>
<tr>
<td>Q</td>
<td>ENDM not within a Macro definition.</td>
</tr>
<tr>
<td>R</td>
<td>Expression stack overflow, or improper Macro name.</td>
</tr>
<tr>
<td>S</td>
<td>Address out of range (LOAD mode), or improper string termination.</td>
</tr>
<tr>
<td>T</td>
<td>Symbol table overflow.</td>
</tr>
<tr>
<td>U</td>
<td>Undefined variable.</td>
</tr>
<tr>
<td>V</td>
<td>Value is too large for field, has undefined variable, is missing, is illegal type, or END pseudo-op is within a Macro definition.</td>
</tr>
<tr>
<td>W</td>
<td>MAC pseudo-op is within a Macro definition.</td>
</tr>
<tr>
<td>X</td>
<td>Improper index tag, or improper external name.</td>
</tr>
</tbody>
</table>
APPENDIX H

ASSEMBLER IMPROVEMENTS - DISK REVISIONS 3 & 4

This appendix details additions and improvements to the assembly language introduced on master disk revisions 3 and 4. Information in this appendix was obtained from Prime internal memos PE-TN-47 and PE-TN-50.
PMA IMPROVEMENTS

The following improvements have been made in PMA for the Rev. 3 Master Disk:

**SUBR/ENT Logic**

PMA formerly truncated the internal name of an entry point to four characters when looking up the value for the entry point. The new SUBR logic will first search for the name as given, and then, if it was not found, truncate it to four characters and search again. Because of this change, symbol references in SUBR/ENT statements will not be included in the concordance.

**Multi-Word Literals**

Literal expressions may now be multi-word data items. For example:

```
DAC = 'C'012345678'
DAC = 2.51E4
```

**PCVH Pseudo-Op**

The PCVH pseudo-op directs the assembler to print symbol values in the concordance in hexadecimal instead of octal.

**Phase Error Detection**

The assembler will now flag phase errors (a symbol having a different value in pass 2 than in pass 1) with a 'Y' error diagnostic.

**XSET Pseudo-Op**

The XSET pseudo-op is functionally equivalent to the SET pseudo-op, except that symbols defined with XSET will not be included in the concordance.

**B and Y Register Attributes**

The initial value of the B and X registers at the start of an assembly are now available as attribute numbers 101 and 102 respectively.
The following loader control pseudo-ops have been added to PMA:

**D16S - Desector in 16K Sectored Mode**

The D16S pseudo-op directs the loader to enter 16K sectored desectorization mode. It is equivalent to the LXD pseudo-op.

**D32S - Desector in 32K Sectored Mode**

The D32S pseudo-op directs the loader to enter 32K sectored desectorization mode. It is equivalent to the EXD pseudo-op.

**D32R - Desector in 32K Relative Mode**

The D32R pseudo-op directs the loader to enter 32K relative desectorization mode.

**D64R - Desector in 64K Relative Mode**

The D64R pseudo-op directs the loader to enter 64K relative desectorization mode.

**SDM - Set Default Desectorization Mode**

The SDM pseudo-op directs the loader to set its default desectorization mode to the mode defined by the expression in the variable field of the SDM pseudo-op. Legal values of the expression are:

<table>
<thead>
<tr>
<th>Value</th>
<th>Mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>16K Sectored Mode</td>
</tr>
<tr>
<td>1</td>
<td>32K Sectored Mode</td>
</tr>
<tr>
<td>2</td>
<td>64K Relative Mode</td>
</tr>
<tr>
<td>3</td>
<td>32K Relative Mode</td>
</tr>
</tbody>
</table>

The SDM pseudo-op does not change the current desectorization mode.
DDM - Deselect in Default Desectorization Mode

The DDM pseudo-op directs the loader to enter the desectorization mode defined by its default desectorization mode. The default desectorization mode is initially set at the start of a load and is only changed by a SLM pseudo-op.
The following PMA extensions have been implemented for the Rev 4 master disk:

**New Constant Forms**

The following new constant forms are now processed:

1. **Binary Constants**

   A percent sign followed by a string of binary digits or the characters B' followed by a binary digit string followed by a ' will be processed as a binary constant.

   **Examples:**
   
   \[ \text{B'101'} = 5 \]
   \[ \text{\$11011} = '33 \]

2. **Single Character Constants**

   The form R'c', where c is any character, will be processed as the character code of c.

   **Examples:**
   
   \[ R'A' = '301 \]
   \[ R'' = '247 \]

**C64R (Check 64K Relative) Pseudo-Op**

The C64R pseudo-op directs the assembler to flag (with an 'S' diagnostic) any instruction that is incompatible with the 64K relative addressing mode. The following cases are detected:

1. **An indirect DAC**

2. **An indirect single word memory reference instruction whose address is not in either sector zero or within the relative reach of the instruction.**
N64R (Not 64K Relative) Pseudo-Op

The N64R pseudo-op directs the assembler to output a flag in the object text to inform the loader that the program is not to be loaded in the 64K relative addressing mode. If such a program is loaded in the 64K relative addressing mode, the loader will report a 'N6' error.

SETB  Pseudo-Op Extension

An additional form of the SETB pseudo-op is now processed to allow the size of the desectorization to be specified. The format is:

```
SETB   exp1, exp2
```

where:

- exp1 - starting address of desectorization area
- exp2 - size of desectorization area

EQU, SET, and XSET  Pseudo-Op Extensions

The EQU, SET, and XSET pseudo-ops will allow the assignment of stack relative and external values to symbols.

DUII (Define UII)  Pseudo-Op

The DUII pseudo-op is used to trigger the loading of a UII package based on the instruction set used by previously loaded code and the hardware available on the machine the program is to execute on. The format is:

```
DUII    exp1, exp2
```

where:

- exp1 - bit mask defining instruction groups that UII package emulates
- exp2 - bit mask defining instruction sets that must be available for the execution of the UII package
The bit assignments for instruction set options are as follows:

- 13 - Double Precision Floating Point
- 14 - Single Precision Floating Point
- 15 - P300 only instructions
- 16 - High Speed Arithmetic

**LIR (Load Is Required) Pseudo-Op**

The LIR is used to trigger the loading of a program based on the instruction groups used by previously loaded code. The format is:

```
LIR  expl
```

where:

- expl - bit mask defining instruction groups that are to trigger loading. Bit assignments are the same as for DUIS.

The program will be loaded if any of the instruction groups specified have been used in previously loaded code.

**CENT (Conditional Entry) Pseudo-Op**

The CENT pseudo-op is equivalent to the ENT pseudo-op except that the loader will only process it if the decision to load a module containing a CENT pseudo-op had been made prior to the occurrence of the CENT statement.

**DYNM (DYNAMIC) Pseudo-Op**

The DYNM pseudo-op is used to declare stack relative symbols. Since references to stack relative symbols generate two-word instructions, stack relative symbols must be declared before they are used. The format of a DYNM statement is:

```
DYNM  s1, s2, ..., sN
```

where:

- \( si \) = a specifier in one of the following formats:

1) symbol
2) symbol (exp1)
3) symbol = exp2
4) symbol (exp1) = exp2
5) = exp2
In the following descriptions of the formats, the following abbreviations are used:

- sc - current stack allocation count (#106)(initially = 2)
- sm - maximum allocation count (#107)
- symbol - symbol to be assigned stack relative offset
- exp1 - expression defining number of words for symbol
- exp2 - expression defining stack offset

1. symbol
   - symbol is assigned offset = sc
   - sc = sc + 1
   - if (sc .GT. sm) sm = sc

2. symbol (exp1)
   - symbol is assigned offset = sc
   - sc = sc + exp2
   - if (sc .GT. sm) sm = sc

3. symbol = exp2
   - symbol is assigned offset = exp2
   - if (exp2 + 1 .GT. sm) sm = sc

4. symbol (exp1) = exp2
   - symbol is assigned offset = exp2
   - if (exp2 + exp1 .GT. sm) sm = exp2 + exp1

5. = exp2
   - sc = exp2

Index Field Extensions

The index field has been expanded to allow the specification of both indexing and indirection. The possible contents of this field are:

,1 indexed
,* indirect
,1* pre-index, indirect
,*1 indirect, post-index
Indirection may still be specified by an asterisk appended to the op-code.

Expression Evaluation Extensions

The following modifications have been made to the expression evaluator:

1. Stack Pre-Decrement Expression

   A stack pre-decrement expression is an expression consisting only of the characters `-@`. It may only be used in the address expression of a memory reference instruction.

2. Stack Post-Increment Expression

   A stack post-increment expression is an expression consisting only of the characters `@+`. It may only be used in the address expression of a memory reference instruction.

3. Stack Relative Special Symbol

   The symbol `@@` has been the value of a zero offset from the stack base when used in an expression (other than in the preceding two special cases).

4. Resultant Mode of Arithmetic Operations

   All arithmetic operations other than addition and subtraction will give a result mode of absolute. The resultant mode of an addition or subtraction operation depends on the modes of the left and right operands, as shown in the following tables.

<table>
<thead>
<tr>
<th>mode of left operand</th>
<th>+</th>
<th>abs</th>
<th>m*rel</th>
<th>stack + j*rel</th>
</tr>
</thead>
<tbody>
<tr>
<td>abs</td>
<td>abs</td>
<td>m*rel</td>
<td>stack + j*rel</td>
<td></td>
</tr>
<tr>
<td>n*rel</td>
<td>n*rel</td>
<td>(m+n)*rel</td>
<td>stack + (j+n)*rel</td>
<td></td>
</tr>
<tr>
<td>stack + i*rel</td>
<td>stack + i*rel</td>
<td>stack + (i+m)*rel</td>
<td>P</td>
<td></td>
</tr>
</tbody>
</table>

H-9
<table>
<thead>
<tr>
<th>mode of left operand</th>
<th>mode of right operand</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>abs</td>
</tr>
<tr>
<td>abs</td>
<td>abs</td>
</tr>
<tr>
<td>n*rel</td>
<td>n*rel</td>
</tr>
<tr>
<td>stack +</td>
<td>stack +</td>
</tr>
<tr>
<td>i*rel</td>
<td>i*rel</td>
</tr>
</tbody>
</table>

**NOTES:**
- P = Prohibited
- 1*rel = rel
- 0*rel = abs

5. Resultant Mode of Expression

The resultant mode of an expression must be one of the following:

- Absolute
- 1*rel
- Stack + absolute
- External
- Stack pre-decrement
- Stack post-increment

If the final mode of an expression is not one of the above, a 'Z' diagnostic will be reported. Also, the result mode must be one consistent with its usage. For example, if a relocatable expression appeared in the address field of a BSS statement, a 'Z' error would be reported since BSS cannot correctly process a relocatable value.

**Support of New Instructions**

All PRIME 300 and floating point instructions will be processed by the assembler.
Generation of Special Relative Address Forms

A special relative address form will be generated for a memory reference instruction if any of the following conditions are met:

1. The address is stack relative.
2. The address is stack pre-decrement (\(\@\)).
3. The address is stack post-increment (\(\@+\)).
4. If the instruction is pre-indexed (1\(^*\)) with a non-absolute address, or an absolute address \(\geq '100\).
5. The instruction is post-indexed (\(*1\)) with an absolute address \(\leq '100\).
6. If the instruction does not have a non-special relative form (non-zero op-code extension).
7. A percent sign (\%) is appended to the op-code.

Assembly Listing Changes

The following changes have been made in the assembly listing:

1. All addresses are printed as six digits.
2. All instruction (and address constant) addresses have a character appended to the end to indicate the mode of the address. The following characters are used:

   A - Absolute
   space - Relocatable
   E - External
   S - Stack Relative

New Object Text Generation

The object text generated by the assembler is in a new format only accepted by the Rev 4 (and subsequent) loaders.
EXAMPLES OF REV. 4 PMA EXTENSIONS:

REL

* NEW CONSTANT FORMS *

DATA B'11100001' BINARY CONSTANT

DATA X'1110001' SAME CONSTANT, DIFFERENT FORM

DATA R'C',R'_' SINGLE CHARACTER CONSTANTS

DATA 0 * C64R PSEUDO-OP *

C64R

DAC# 0 FLAG INDIRECT DAC

LDA# **+257 INDIRECT, OUT OF RELATIVE REACH

SETB BASE.25
DUII '14,1  DUII FOR A UII PACKAGE THAT SATISFIES
SINGLE AND DOUBLE PRECISION FLOATING POINT AND
REQUIRES HIGH SPEED ARITHMETIC FOR EXECUTION

LIR %1180  LOAD IF SINGLE AND/OR DOUBLE FLOATING POINT USED

DYNM ADDRESS, NAME(3), BUFFER(80), TEMP1, TEMP2
000003 (0071) DYNM ABC=3, DEF(4)=5
000005
000024 (0072) DYNM =T0, T1, T2, T3
000024
000025
000026
000041: (0083) LOCAL BSS 1
(0085) *
(0082) *
(0083) *
(0081) *
(0084) LDA LOCAL
(0085) LDA LOCAL, 1
(0086) LDA# LOCAL
(0087) LDA# LOCAL, 1
(0088) *
(0089) *
(0090) *
(0091) LDA# LOCAL
(0092) LDA @+3
(0093) LDA @+
(0094) LDA @-
(0095) *
(0096) LDA# LOCAL, 1
(0097) LDA NAME+3, 1
000055: (0098) LDA @+, @+1
000057: (0099) LDA @-, @-1
000060: (0100) *
000062: 02. 000041
000063: 22. 000041
000064: 42. 000041
000065: 62. 000041

FIRST, THE FOUR NON-SPECIAL RELATIVE FORMS

THE 16 SPECIAL RELATIVE FORMS

'X' REQUIRED BECAUSE SHORT FORM AVAILABLE

(NAME WAS DECLARED IN A DYNM)
LDA% LOCAL,*
LDA TEMP1,*
LDA @+
LDA -*,*
LDA LOCAL,1* PREINDEXED, >= '100
LDA ADDRESS,1*
LDA '30,*1 POST-INDEXED, <'100
LDA ADDRESS,*1

 resultList is 1*REL, OK
 resultList is 2*REL, ERROR
 resultList is stack + abs, OK
 resultList is stack + rel, error
 cannot subtract stack relative
 cannot add 2 stack relatives
<table>
<thead>
<tr>
<th></th>
<th>Address 1</th>
<th>Address 2</th>
<th>Address 3</th>
<th>Address 4</th>
<th>Address 5</th>
<th>Address 6</th>
<th>Address 7</th>
<th>Address 8</th>
<th>Address 9</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABC</td>
<td>0000035</td>
<td>0071</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ADDRESS</td>
<td>0000025</td>
<td>0068</td>
<td>0107</td>
<td>0109</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BASE</td>
<td>000006</td>
<td>0015</td>
<td>0030</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BUFFER</td>
<td>000005</td>
<td>0060</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DEF</td>
<td>000005</td>
<td>0071</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LOCAL</td>
<td>000041</td>
<td>0030</td>
<td>0034</td>
<td>0035</td>
<td>0036</td>
<td>0037</td>
<td>0001</td>
<td>0006</td>
<td>0101</td>
</tr>
<tr>
<td>NAME</td>
<td>0001035</td>
<td>0068</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>T1</td>
<td>0000245</td>
<td>0072</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>T2</td>
<td>0000255</td>
<td>0072</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>T3</td>
<td>0000265</td>
<td>0072</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TEMP1</td>
<td>0001265</td>
<td>0068</td>
<td></td>
<td>0102</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TEMP2</td>
<td>0001275</td>
<td>0068</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

0006 ERRORS (PMA-1080.011)
COMM PSEUDO-OP (FORTRAN COMPATIBLE COMMON)

This pseudo-op is for definition of FORTRAN-compatible named COMMON areas, which are defined downward starting from the highest location occupied by the loader version in use during actual loading. The syntax is:

\[
\text{Label} \quad \text{COMM} \quad S_1, S_2, \ldots, S_n
\]

where 'Label' is the name of the common block and each 's' is a specifier in one of the formats defined for the DYNM pseudo-operation. 'Label' assigns a name to the block as a whole and each 's' specifies named variables or arrays within the block. Two counters are maintained on a per common block basis - a current allocation count and a maximum allocation count - as in DYNM:

<table>
<thead>
<tr>
<th>Counter</th>
<th>Initial Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>sc</td>
<td>0</td>
</tr>
<tr>
<td>sm</td>
<td>0</td>
</tr>
</tbody>
</table>

Additional COMM statements with the same block name are treated as continuations of the earlier block.

RLIT PSEUDO-OP (LITERALS OPTIMIZED FOR RELATIVE MODES)

RLIT is a specification-type pseudo-operation that directs the assembler to handle literals in a way that is optimized for relative addressing modes. Normally (i.e., without RLIT), literals are assigned locations following a FIN or END statement. If a defined literal is referenced following a FIN, it is assigned another location following the next FIN or END statement. However, in a program that is proceeded by RLIT, a literal that has already been defined and is still within the relative or multiword reach is referenced directly. (A new location is not allocated.)
INDEX

#COPY COMMAND 6-2
#DONE COMMAND 6-3
#INSERT COMMAND 6-1
#UPDATE COMMAND 6-2

ABSOLUTE (ABS) MODE 1-16
ADDRESS RF SOLUTION 1-16
ADDRESSES, SYMBOLIC 3-13
ADDRESSING, EXTENDED 1-17

ARGUMENT IDENTIFIERS, MACROS 5-7
ARGUMENT REFERENCES, MACROS 5-3
ARGUMENT SUBSTITUTION, MACROS 5-4
ARGUMENT VALUES, MACROS 5-4

ASCII CHARACTER CODES E-1
ASCII CONSTANTS 2-6
ASCII STRINGS 4-11

ASSEMBLER ATTRIBUTE REFERENCES, MACROS 5-8
ASSEMBLER/LOADER INTERACTION 1-16
ASSEMBLY CONTROLLING PSEUDO OPERATIONS 4-4
ASSEMBLY LANGUAGE, BASIC ELEMENTS 1-3
ASSEMBLY LANGUAGE, EXAMPLE OF STATEMENTS 1-5
ASSEMBLY LANGUAGE, FEATURES 1-2

ASTERISK, DOUBLE (INITIALLY ZERO) 3-7
ASTERISK, SINGLE (CURRENT LOCATION) 3-7
ASTERISK, SINGLE (INDIRECT ADDRESSING) 3-13
ASTERISK, TRIPLE (DUMMY INSTRUCTION) 3-13

ATTRIBUTE REFERENCES, ASSEMBLER (MACROS) 5-8

R AND BB NOTATION (DATA CONSTANTS) 4-13

BINARY SCALING (B) NOTATION 4-15

BIT REFERENCE INSTRUCTIONS 3-18

CALLS, MACROS 5-3

CHANGE PAGE HEADING LINES 2-3

COMMENTS 2-3

CONDITIONAL ASSEMBLY PSEUDO-OPERATIONS 4-44

CONSTANTS 1-8

CONSTANTS, DEFINED 2-5

CONSTANTS, NUMERICAL 4-12

CONSTANTS, REPEATED 4-21

CROSS REFERENCE LISTING 1-15

DATA DEFINING PSEUDO OPERATIONS 4-11

DATA STATEMENTS, MULTIPLE AND IMPLIED 4-21

DECIMAL CONSTANTS 2-5

DESSECTORIZING 1-16

DEVICE CODES, INPUT/OUTPUT DEVICES D-1

DOUBLE ASTERISK (INITIALLY ZERO) 3-7

DOUBLE PRECISION FIXED POINT 4-16

DOUBLE PRECISION FLOATING POINT *4-18

DUMMY WORDS, MACROS 5-5

R AND EE NOTATION (DATA CONSTANTS) 4-13

EQUALS SIGN (LITERALS) 3-9

ERROR MESSAGES, ASSEMBLER G-1

X-1
INDEX (Cont'd)

MACROS, ARGUMENT VALUES 5-4
MACROS, CALLS 5-3
MACROS, DUMMY WORDS 5-5
MACROS, LISTING AND ASSEMBLY CONTROL 5-10
MACROS, LOCAL REFERENCES 5-9
MAGNITUDES OF CONSTANTS 2-5
MAP, MEMORY 1-18
MEMORY REFERENCE INSTRUCTIONS 3-11
MNEMONICS, INSTRUCTION 3-2
MULTIPLE DATA STATEMENTS 4-21
MULTIPLE STATEMENTS PER LINE 2-3
OBJECT OUTPUT OF ASSEMBLER 1-12
OCTAL CONSTANTS 2-5
OPERATION FIELD, INSTRUCTION STATEMENTS 3-1
OPERATION FIELD, MEMORY REFERENCE INSTRUCTIONS 3-13
OPERATORS (ARITHMETIC, LOGICAL, RELATIONAL AND SHIFT) 2-8
POWERS OF 10 (E) NOTATION 4-15
PRIORITY, OPERATORS 2-9
PROGRAM LINKING PSEUDO-OPERATIONS 4-38
PSEUDO-OPERATIONS 4-1
PSEUDO-OPERATIONS, *** 4-31
     ABS 4-4
     ASSEMBLY CONTROLLING 4-4
     BACK/BACK TO 5-13
     BCI 4-28
     BES 4-34
     BSS 4-34
     RSZ 4-34
     CALL 4-39
     CF1-CF5 4-7
     COMM 4-36
     CONDITIONAL ASSEMBLY 4-44
     DAC 4-29
     DATA 4-11
     DATA DEFINING 4-11
     DBP 4-24
     DEC 4-23
     EJCT 4-8
     ELSE 4-46
     END 4-6
     ENDC 4-46
     ENDM 5-3
     ENT 4-40
     EQU 4-32
     EXD 4-9
     EXT 4-38
INDEX (Cont'd)

PEEUDO-OPERATIONS  4-47
FIN  4-6
GO/GO TO  4-7
HEX  4-26
IF  4-44
IFM/IFF/IFZ/IFN  4-45
INTRODUCTION  1-10
LIST  4-8
LISTING CONTROL  4-8
LOADER CONTROLLING  4-9
LSMD  5-11
LSTM  5-10
LXD  4-9
MAC  5-2
MACRO DEFINITION AND CALLS  5-2
MACRO LISTING AND ASSEMBLY CONTROL  5-10
MOR  4-6
NLSM  5-12
NLST  4-8
OCT  4-25
ORG  4-4
PROGRAM LINKING  4-38
REL  4-4
SAY  5-13
SET  4-32
SETB  4-9
SETC  4-35
STATEMENT FORMAT
STORAGE ALLOCATION  4-34
SUBR  4-40
VARIABLE (SYMBOL) DEFINING  4-32
VFO  4-27
XAC  4-30

RDALN  6-1
REFERENCE DOCUMENTS  1-1
RELATIONAL OPERATORS  2-10
RELOCATABLE (REL) MODE  1-16
SCOPE OF HANDBOOK  1-1
SHIFT INSTRUCTIONS  3-16
SHIFT OPERATORS  2-10
SIGN CONVENTIONS  2-12
SINGLE PRECISION FIXED POINT  4-12
SINGLE PRECISION FLOATING POINT  4-18
SOURCE FILE MERGING COMMANDS  6-1
SPACE CONVENTIONS, OPERATORS  2-9
STATEMENTS  2-1
STATEMENTS, CONTINUED  2-3
STORAGE ALLOCATION PSEUDO-OPERATIONS  4-34
STRINGS, ASCII  4-11
SUBROUTINES, LOADING  1-17

X-4
INDEX (Cont'd)

SYMBOL (VARIABLE) DEFINING PSEUDO-OPERATIONS  4-32
SYMBOL CROSS REFERENCE LISTING  1-15
SYMBOLIC ADDRESSES  3-13
SYMBOLIC INSTRUCTION  1-6
SYMBOLIC INSTRUCTION, INTERPRETATION OF  1-7
SYMBOLIC NAMES  1-9
TWO-PASS ASSEMBLY  1-12
VARIABLE (SYMBOL) DEFINING PSEUDO OPERATIONS  4-32
VARIABLE FIELD, INSTRUCTION STATEMENTS  3-7
VARIABLE FIELD, MEMORY REFERENCE INSTRUCTIONS  3-13
VARIABLES  1-8