MAN1673

MACRO ASSEMBLER
User Guide

Revision A
May 1975

PRIME
Computer, Inc.
145 Pennsylvania Ave.
Framingham, Mass. 01701

First Printing January 1973
Revision A April 1975

MAN 1673

Performance characteristics are
subject to change without notice.

CONTENTS

ILLUSTRATIONS

TABLES

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

PRIME 200 ASSEMBLY LANGUAGE

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    ^3000
      CRA
C     OTA    ^1720              DISPLAY A REG IN PANEL IND
      SR1
      JMP*   DOS
A     IRX
      JMP    A
B     IRX
      JMP    B
      AOA                       DATA INDICATORS COUNT SLOWLY
      FAIL                      DELIBERATE ERROR LINE
      JMP    C
DOS   OCT    30000
      END
END
```

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.

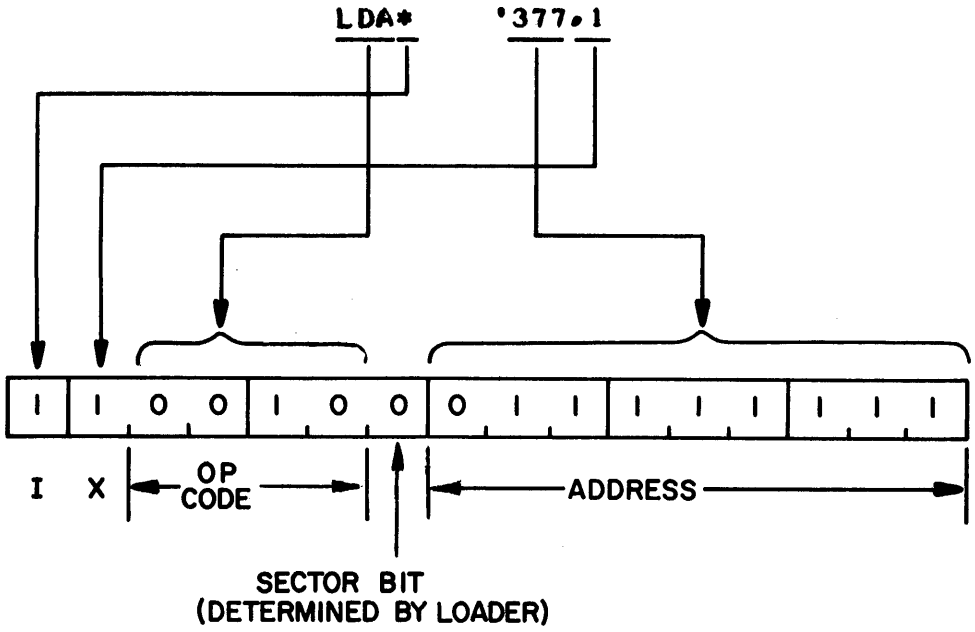


Figure 1-2. Interpretation of Symbolic Instruction.

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:

'123	-'777	'-1777	Octal
$89AB	-$FFFF	$-00FF	Hexadecimal
1234	-9999	32767	Decimal

The constant forms shown above are all single-precision (ie. 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:

1.23BB6	1.23EE2BB6	Fixed Point Double Precision
1.23E2	1.1092	Floating Point Single Precision
1.23EE2		Floating Point Double Precision

The assembler also accepts ASCII constants:

C'A'	(The letter A, left justified in a 16-bit word)
C'AB'	(The letters A and B packed into a 16-bit word.)

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.

```
                    (0001) *SHORT DEMO PROGRAM EXAMPLE
                    (0002) *
                    (0003) *
                    (0004)        ABS
                 003000 (0005)        ORG    ^3000      DISPLAY A REG IN PANEL IND
    03000:     140040 (0006)        CRA
    03001:     171720 (0007) C      OTA    ^1720
    03002:     100072 (0008)        SR1
    03003:  41 03012  (0009)        JMP*   DOS
    03004:     140114 (0010) A      IRX
    03005:  01 03004  (0011)        JMP    A
    03006:     140114 (0012) B      IRX                DATA INDICATORS COUNT SLOWLY
    03007:  01 03006  (0013)        JMP    B           DELIBERATE ERROR LINE
    03010:     141206 (0014)        ADA
F                   (0015)        FAIL
    03011:  01 03001  (0016)        JMP    C
    03012      030000 (0017) DOS    OCT    30000
               003013 (0018)        END


   A            003004   0010  0011
   B            003006   0012  0013
   C            003001   0007  0015
   DOS          003012   0009  0017

0001 LINES WITH ERRORS.       (VERSION P.01)
```

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 16K sectored 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

```
*START 01000     *HIGH  03014     *NAMES 20035     *COMN  23777
*PBRK  03014     *BASE  00100     LIST   00001
```

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.

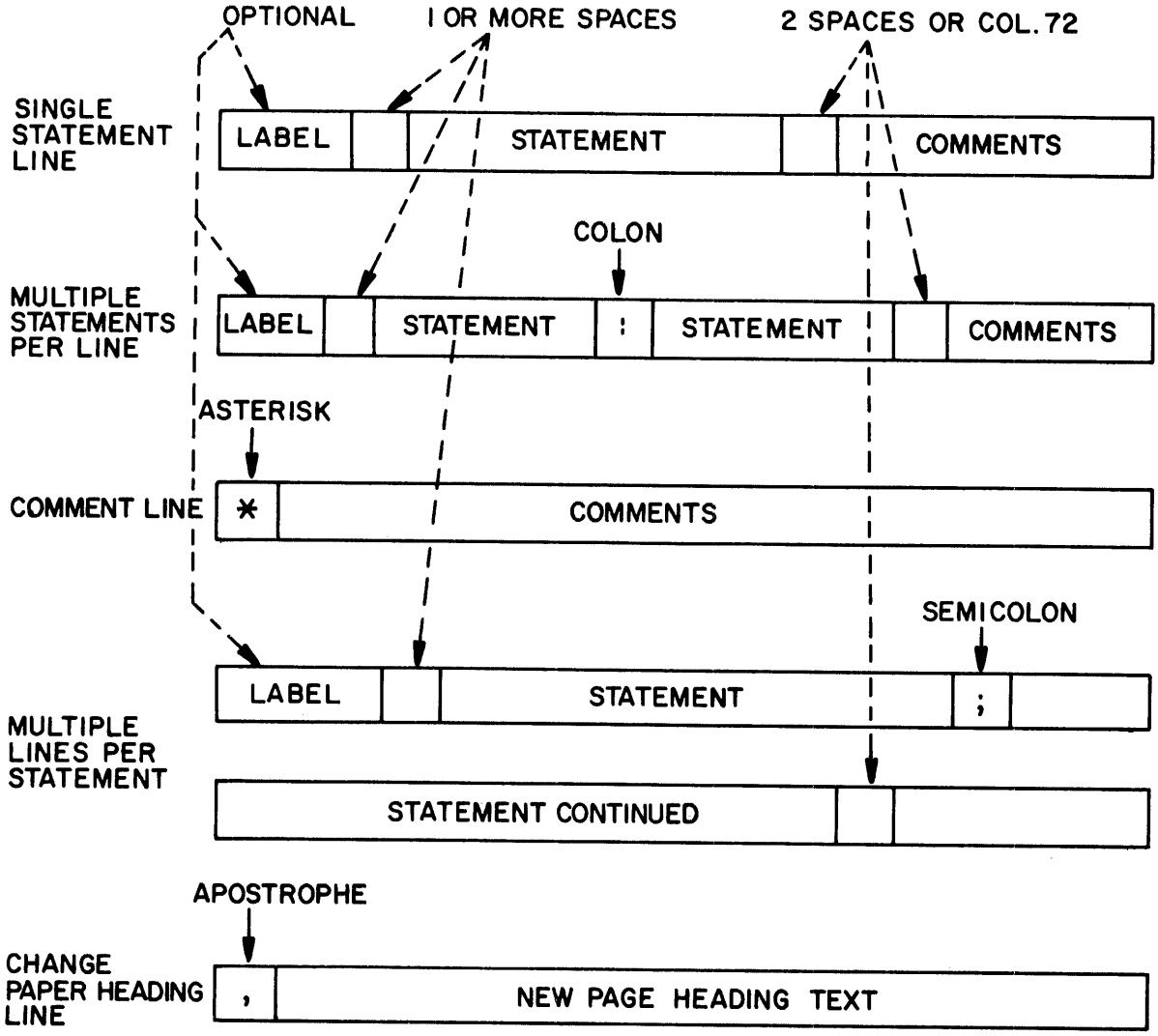


Figure 2-1. Source Input Line Formats

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.

```
                    (0197) *-----TEST OF LONG LABELS.
01252: 02 01215     (0198) ABCDEFGHIJ#KLMNOPQRSTUVWXYZ0123456789 LDA ALPA
01253: 02 01252     (0199)       LDA    ABCDEFGHIJ#KLMNOPQRSTUVWXYZ01234567789
                    (0200) *
                    (0201) *-----TEST OF FREE FORMAT AND LINE CONTINUATION
                    (0202) AB;
                    (0203)   C      A;
01254: 06 01215     (0204)            DD      ALPA.
01255:    000001    (0205)                              1
01256: 01 01215     (0206) AB JMP ALPA
01257:    140114    (0207)       IRX                              COMMENT
01260:    041166    (0208)  LLS 10
01261: 26 01220     (0209)       ADD    BETA.1
01262:    041067    (0210)       LLL    9    COMMENT
                    (0211) *
                    (0212) *-----TEST OF MULTIPLE INSTRUCTIONS PER LINE.
01263: 02 00001     (0213) ABX  LDA    1:    ADD    2:    STA    3
01264: 06 00002
01265: 04 00003
01266:    140040    (0214)  CRA:IAB:CRA:STA 0
01267:    000201
01270:    140040
```

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:

Type	Max. Negative	Max. Positive
Decimal	-32768	+32767
Octal	-'100000	+'777777
Hexadecimal	-$4000	+$7FFF

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:

Hexadecimal Digit	Decimal Value
0-9	0-9
A	10
B	11
C	12
D	13
E	14
F	15

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:

```
0'0000  140640 (0004) P2$6      DATA    C'A'
0'0001  140732 (0005)           DATA    C'AZ'
```

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 DAC 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.

```
02002: 02.01234 (0006) P2$7      LDA    VARI35$
02003: 02.01333 (0007)           LDA    VARI35$+'77
          001234 (0008) VARI35$  SET    '1234
02004: 01.01232 (0009)           JMP    VARI35$-2
```

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:

```
0005  02 0016  (0010)  P2#0      LDA     K100+3
0006  04 0314  (0011)            STA     *+VAR135#-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.

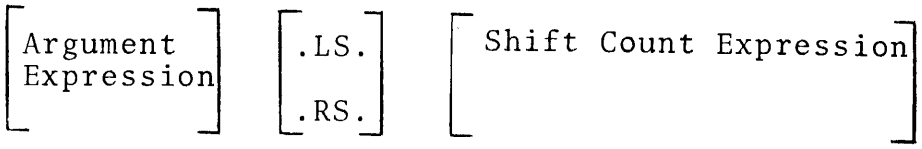
Operator Priority: In expressions with more than one operator, the order of evaluation is governed by operator priority. The operator with the highest priority is performed first. In cases of equal priority, the evaluation proceeds from left to right. Parentheses may be used to alter the natural order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

Priority	Operator(s)
Highest	* /
	+ -
	.RS. .LS.
	.GT. .GE. .EQ. .NE. .LE. .LT.
	.AND.
	.OR.
Lowest	.XOR.

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:

```
                    (0234) +----TEST RELATIONAL OPERATORS
01304  000001 (0235)       DATA  5 .EQ. 5, 5 .EQ. 6, 6 .EQ. 5
01305  000000
01306  000000
01307  000000 (0236)       DATA  5 .NE. 5, 5 .NE. 6, 6 .NE. 5
01310  000001
01311  000001
01312  000000 (0237)       DATA  5 .GT. 5, 5 .GT. 6, 6 .GT. 5
01313  000000
01314  000001
01315  000000 (0238)       DATA  5 .LT. 5, 5 .LT. 6, 6 .LT. 5
01316  000001
01317  000000
01320  000001 (0239)       DATA  5 .GE. 5, 5 .GE. 6, 6 .GE. 5
01321  000000
01322  000001
01323  000001 (0240)       DATA  5 .LE. 5, 5 .LE. 6, 6 .LE. 5
01324  000001
01325  000000
```

Shift Operators: The shift operators perform a logical right or left shift of an expression, using the syntax:



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

Examples:

```
                (0241) *
                (0242) *------TEST SHIFT OPERATOR
01326:   000101 (0243)       DATA  ^1010 .RS. 3, ^1010 .LS. 3, ^1010 LS. 4
01327:   010100
01330:   020200
01331:   010100 (0244)       DATA  ^1010 .RS. -3, ^1010 .LS. -3
01332:   000101
                (0245) *
```

Logical Operators: Examples are shown below.

```
                    (0229) *-----TEST LOGICAL OPERATORS
01301:    001101    (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:

```
02007: 02.01715 (0013) P2$12     LDA     ZILCH-$3E
02010: 02.01715 (0014)           LDA     ZILCH+$-3E
02011: 02.01755 (0015)           LDA     ALPHA-'37
02012: 02.01755 (0016)           LDA     ALPHA+'-37
02013:    000000 (0017) ZILCH    DEC     0
02014:    000147 (0018) ALPHA    DATA    99
```

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:

```
01242: 02 00012 (0181)      (LDA)     10
01243: 42 00012 (0182)      (LDA*)    10
01244: 42 00012 (0183)      (LDA)*    10
01245: 62 00012 (0184)      (LDA)*    10,1
```

Table 3-1. Summary of PRIME 200 Instruction Codes

CLASS	OP CODE	MNEMONIC	DEFINITION
		REGISTER OPERATE	
G	140040	CRA	CLEAR A
G	140014	CRB	CLEAR B
G	140010	CRL	CLEAR LONG (A AND B)
MR	02	LDA	LOAD A
MR	04	STA	STORE A
MR	15	LDX	LOAD INDEX (ASSEMBLER SETS INDEX BIT)
MR	15	STX	STORE INDEX (ASSEMBLER CLEARS INDEX BIT)
MR	13	IMA	INTERCHANGE MEMORY AND A
G	000201	IAB	INTERCHANGE A AND B
G	140104	XCA	TRANSFER A TO B AND CLEAR A
G	140204	XCB	TRANSFER B TO A AND CLEAR B
G	000111	CFA	COMPUTE EFFECTIVE ADDRESS
		ARITHMETIC	
MR	06	ADD	ADD MEMORY TO A
MR	07	SUB	SUBTRACT MEMORY FROM A
G	141206	A0A (A1A)	ADD ONE TO A
G	140304	A2A	ADD TWO TO A
G	140110	S0A (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

Table 3-1 (Cont)

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		SCB	SET C-BIT
G	140200		RCB	RESET C-BIT
G	000005		SGL	ENTER SINGLE PRECISION MODE
G	000007		DBL	ENTER DOUBLE PRECISION MODE
G	000503		EMCM	ENTER MACHINE CHECK MODE
G	000501		LMCM	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	

MR	03	ANA	AND TO A
MR	05	ERA	EXCLUSIVE OR TO A
G	140401	CMA	COMPLEMENT A
G	140413	LEQ	CONVERT A=0 TO TRUE
G	140412	LNE	CONVERT ^(A=0) TO TRUE
G	140411	LLE	CONVERT A<=0 TO TRUE
G	140414	LGE	CONVERT A>=0 TO TRUE
G	140410	LLT	CONVERT A<0 TO TRUE
G	140415	LGT	CONVERT A>0 TO TRUE
		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	

SH	0414NN	ALL (LGL)	A LEFT LOGICAL
SH	0404NN	ARL (LGR)	A RIGHT LOGICAL
SH	0416NN	ALR	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	LLS	LONG LEFT SHIFT

Table 3-1 (Cont)

SH	0401NN	LRS	LONG RIGHT SHIFT
G	000101	NRM	NORMALIZE
G	000041	SCA	TRANSFER SHIFT COUNTER TO A

BYTE MANIPULATION

G	141340	ICA	INTERCHANGE BYTES OF A
G	141140	ICL	INTERCHANGE BYTES OF A AND CLEAR LEFT BYTE
G	141240	ICR	INTERCHANGE BYTES OF A AND CLEAR RIGHT BYTE
G	141050	CAL	CLEAR LEFT BYTE OF A
G	141044	CAR	CLEAR RIGHT BYTE OF A

TRANSFER AND SKIP

MR	01	JMP	UNCONDITIONAL JUMP
MR	10	JST	JUMP TO EA + 1 AND STORE P IN EA
G	100000	SKP	UNCONDITIONAL SKIP
MR	12	IRS	INCREMENT, REPLACE MEMORY AND SKIP
G	140114	IRX	INCREMENT, REPLACE INDEX AND SKIP
G	140210	DRX	DECREMENT REPLACE INDEX AND SKIP
MR	11	CAS	COMPARE A WITH MEMORY
G	140214	CAZ	COMPARE A WITH ZERO
G	100400	SPL (SGE)	SKIP ON A PLUS
G	101400	SMI (SLT)	SKIP ON A MINUS
G	100040	SZE (SEQ)	SKIP ON A ZERO
G	101040	SNZ (SNE)	SKIP ON A NOT ZERO
G	100220	SGT	SKIP ON A GREATER THAN ZERO
G	101220	SLE	SKIP ON A LESS THAN OR EQUAL TO ZERO
G	100100	SLZ	SKIP ON A BIT 16 ZERO
G	101100	SLN	SKIP ON A BIT 16 ONE
BR	101260+N	SASN	SKIP ON A BIT N SET
BR	100260+N	SARN	SKIP ON A BIT N RESET
G	101001	SSC	SKIP ON C-BIT SET
G	100001	SRC	SKIP ON C-BIT RESET
G	101200	SMCS(SPS)	SKIP ON MACHINE CHECK SET
G	100200	SMCR(SPN)	SKIP ON MACHINE CHECK RESET

Table 3-1 (Cont)

G	101036	SSS	SKIP ON ANY OF SENSE SWITCHES 1-4 SET
G	100036	SSR	SKIP ON NONE OF SENSE SWITCHES 1-4 SET
BR	101240+N	SSN	SKIP ON SENSE SWITCH N SET
BR	100240+N	SRN	SKIP ON SENSE SWITCH N RESET

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:

```
COUNT   IRS    ALPHA
        JMP    *-1
         .
         .
         .
        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:

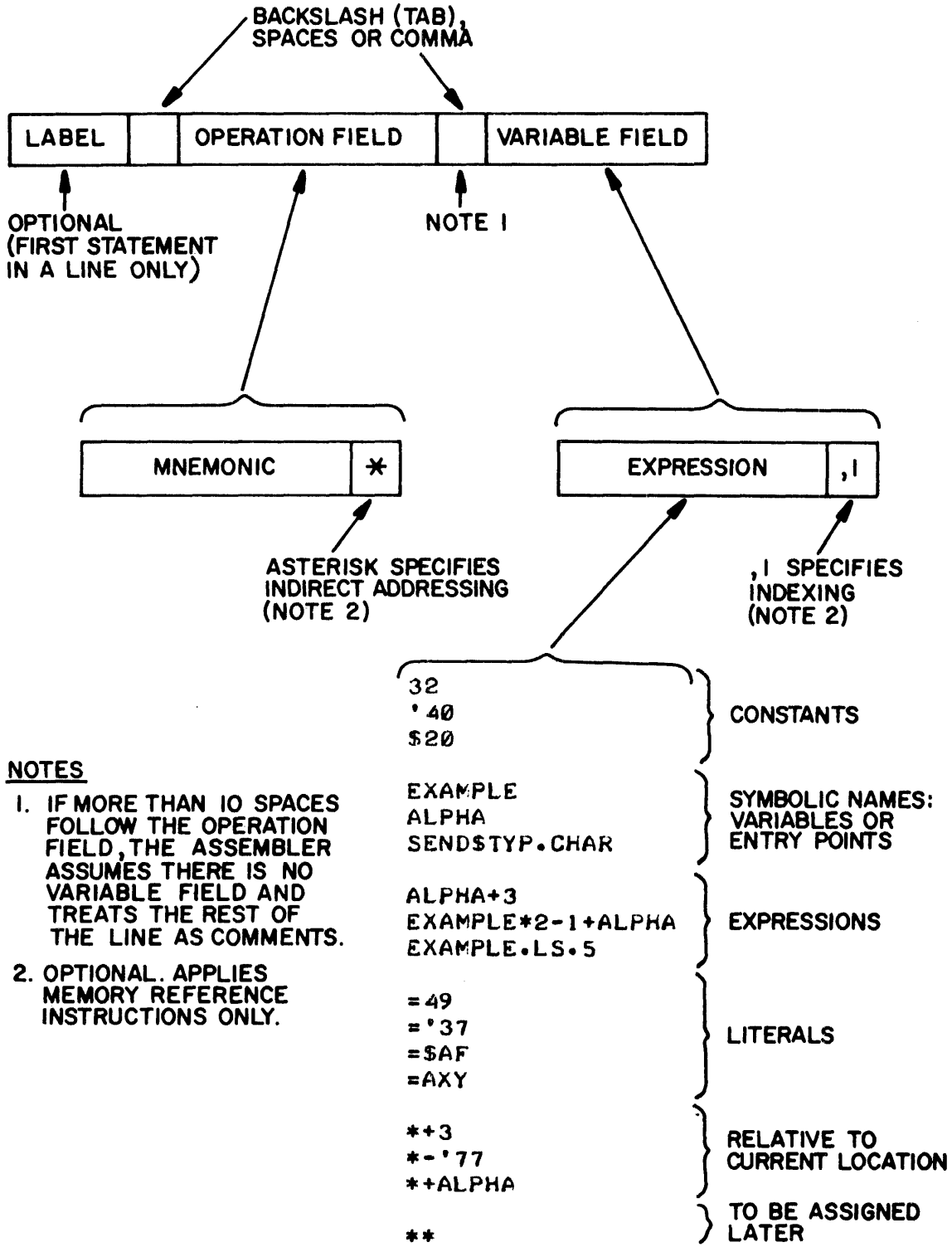


Figure 3-1. General Format of Instruction Statements

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: 06 02004 (0021)          ADD    ='100
02002: 04 02005 (0022)          STA    =X'100'
02003:   000144 (0023)          FIN                          DUMP LITERALS HERE
02004:   000100
02005:   000400

02006: 02 02003 (0024)          LDA    =100                  GENERATE NEW LITERA-
02007: 06 02004 (0025)          ADD    ='100
02010: 04 02005 (0026)          STA    =X'100'
                    (0027)      FIN
02011: 02 02003 (0028)          LDA    =100
02012: 06 02054 (0029)          ADD    =101
02013: 04 02055 (0030)          STA    =X'102


         002054 (0070)          END

02054:   000145
02055:   000402
```

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:

```
02026  02 02033 (0028)          LDA    =C'X'          ASCII DIGIT X, PACKED LEFT
```

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):

```
02027  02 02034 (0029)          LDA    =C'XY'         ASCII DIGITS XY
```

(For ASCII character codes see Appendix E.)

```
02033:  154240
02034:  154331
```

MEMORY REFERENCE INSTRUCTIONS

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

```
                    (0155) *-----MEMORY REFERENCING INSTRUCTIONS.
01215: 01 00100 (0156) ALPA JMP    '100
01216: 02 01215 (0157)      LDA    ALPA
01217: 13 01215 (0158)      ANA*   ALPA
01220: 24 01215 (0159) BETA STA    ALPA,1
01221: 65 01215 (0160)      ERA*   ALPA,1
01222: 06 01215 (0161)      ADD    ALPA
01223: 07 01220 (0162)      SUB    BETA
01224: 10 01224 (0163)      JST    *
01225: 12 01220 (0164)      IRS    BETA
01226: 00 01220 (0165)      PZE    BETA
01227: 00 01220 (0166)      ***    BETA
01230: 11 01230 (0167) GAMA CAS    GAMA
01231: 13 01230 (0168)      IMA    GAMA
01232: 16 00050 (0169)      MPY    40
01233: 17 01220 (0170)      DIV    BETA
01234: 15 01220 (0171)      STX    BETA
01235: 35 01220 (0172)      LDX    BETA
01236: 02 01220 (0173)      DLD    BETA
01237: 04 01220 (0174)      DST    BETA
01240: 06 01220 (0175)      DAD    BETA
01241: 67 00000 (0176)      DSB*   0,1
```

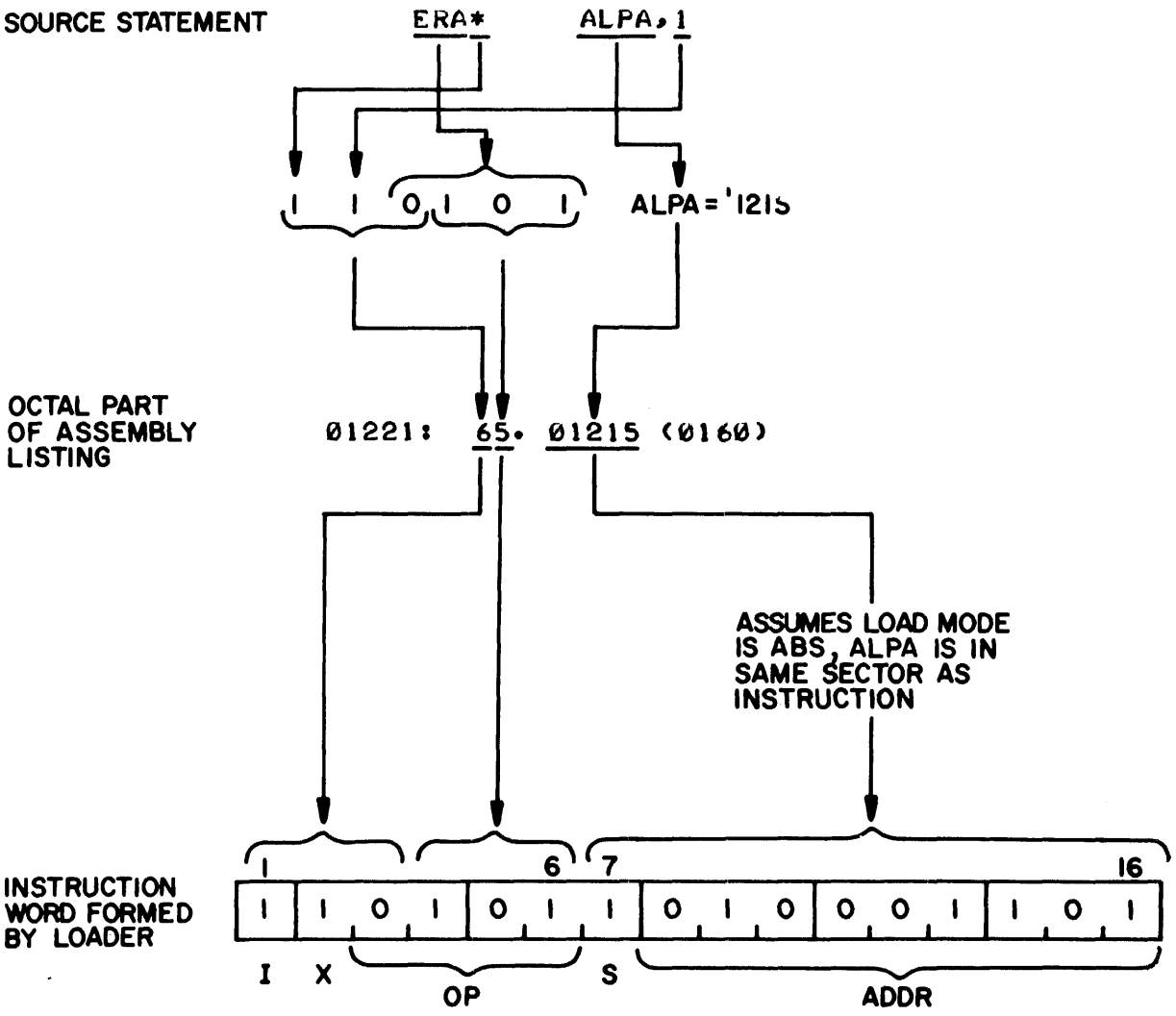


Figure 3-2. Assembly and Loading of Memory Reference Instruction

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 00012 (0187)         ****   10
01247: 00 00012 (0188)         ***    10
01250: 00 00012 (0189)         **     10
01251: 00 00012 (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.

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 (00030) EXAMPLE  LDA     ALPHA,TEST  EQ 5
       000005 (00021) 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:

```
                    (0140) *------I/O INSTRUCTIONS
01204:   030100     (0141)       OCP   '100
01205:   070101     (0142)       SKS   '101
01206:   170102     (0143)       SMK   '102
01207:   130106     (0144)       INA   '106
01210:   170123     (0145)       OTA   '123
                    (0146) *
```

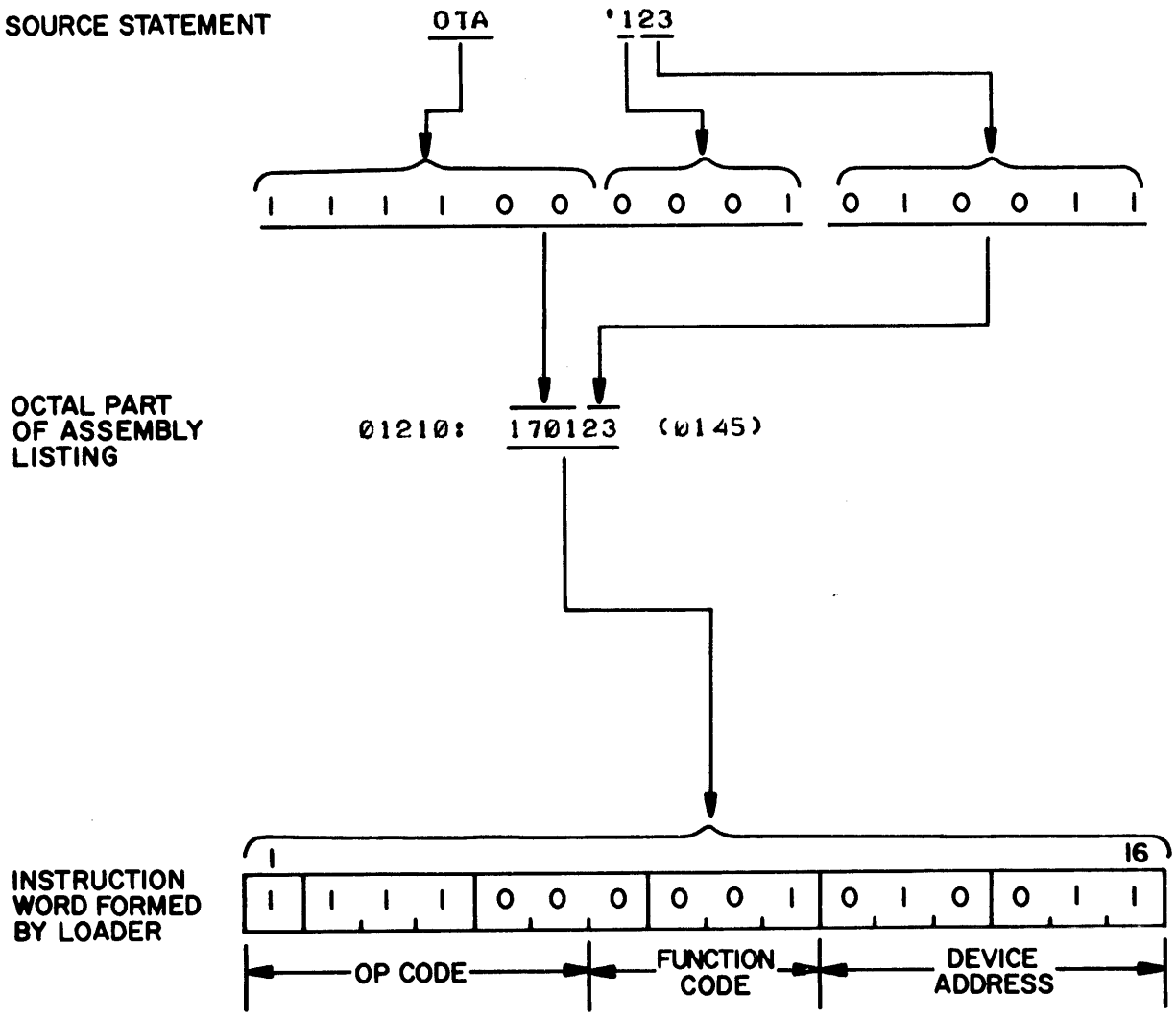


Figure 3-3. Assembly and Loading of Input/Output Instruction

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:

```
                    (0124) *-----SHIFT INSTRUCTIONS
01166:    041400    (0125) ENTR LGL   0
01167:    041400    (0126)      ALL   0
01170:    040577    (0127)      ARS   1
01171:    040676    (0128)      ARR   2
01172:    040475    (0129)      LGR   3
01173:    040475    (0130)      ARL   3
01174:    041574    (0131)      ALS   4
01175:    041673    (0132)      ALR   5
01176:    040072    (0133)      LRL   6
01177:    040171    (0134)      LRS   7
01200:    040270    (0135)      LRR   8
01201:    041067    (0136)      LLL   9
01202:    041166    (0137)      LLS   10
01203:    041265    (0138)      LLR   11
```

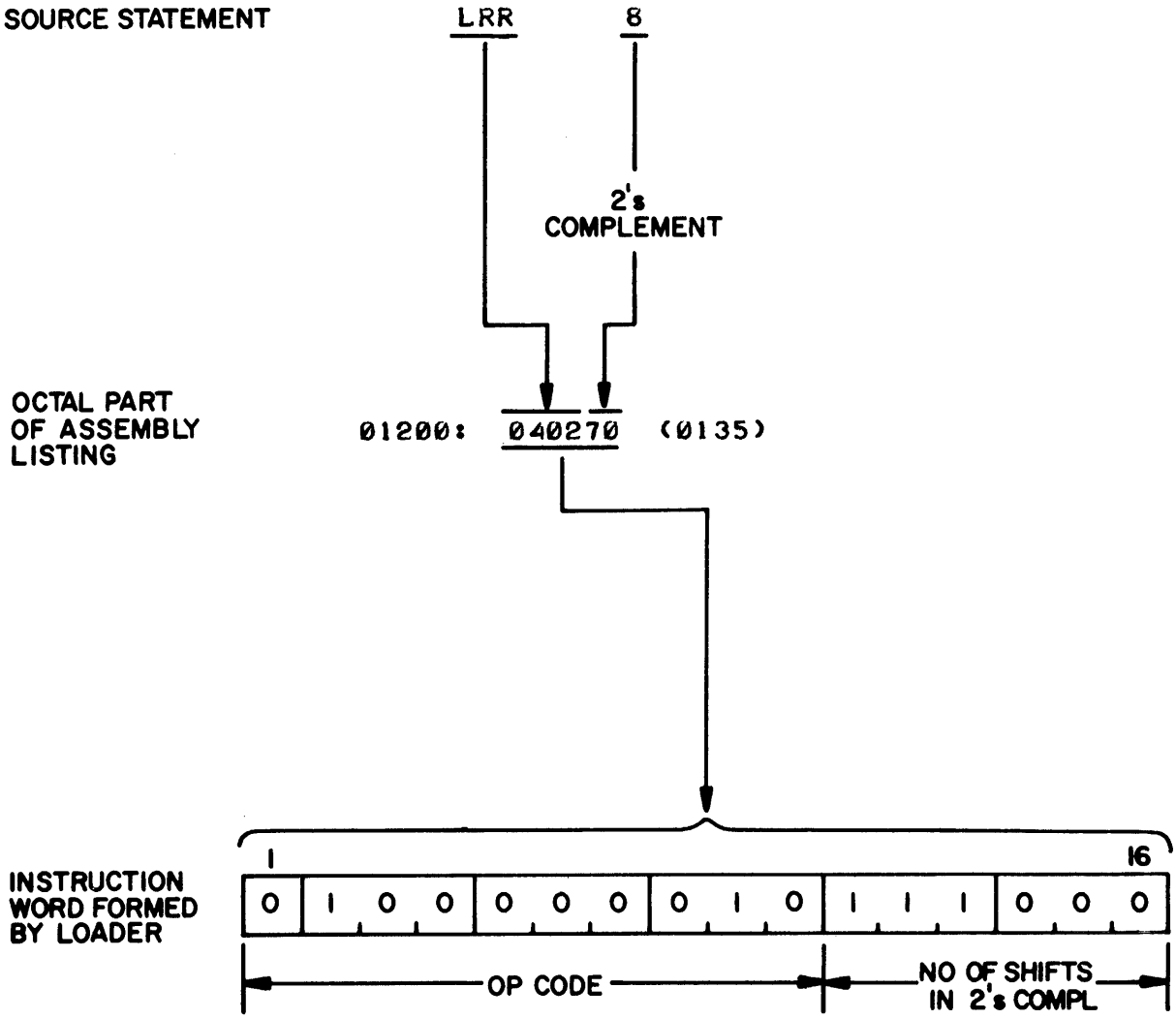


Figure 3-4. Assembly and Loading of Shift Instructions

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:

```
                    (0147) *-----BIT REFERENCE INSTRUCTIONS.
01211:    101240 (0148)       SSN   1
01212:    100241 (0149)       SR    2
01213:    101275 (0150)       SAS   15
01214:    100277 (0151)       SAR   16
```

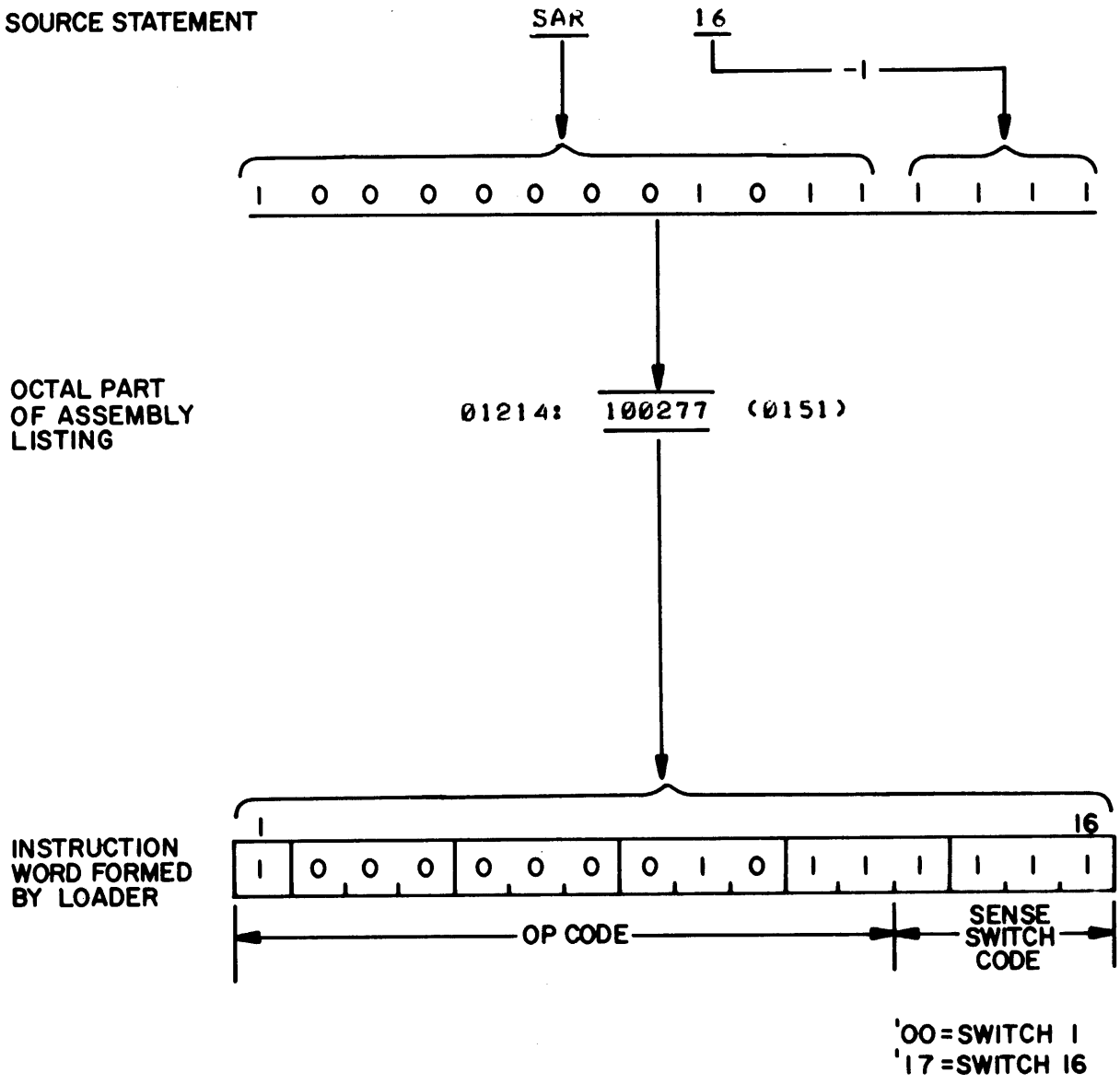


Figure 3-5. Assembly and Loading of
Bit Reference Instructions

GENERIC 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:

```
                          (0015) *-----GENERIC INSTRUCTIONS
01013:    000000  (0016) STRT HLT
01014:    000001  (0017)      NOP
01015:    000005  (0018)      SGL
01016:    000007  (0019)      DBL
01017:    000011  (0020)      DXA
01020:    000041  (0021)      E16S
01021:    000013  (0022)      EXA
01022:    000043  (0023)      E32S


01025:    000021  (0026)      RNP
01026:    000021  (0027)      RNC
01027:    000041  (0028)      SCA
01030:    000043  (0029)      JNK
01031:    000101  (0030)      NRM
01032:    000105  (0031)      EFXM
01033:    000107  (0032)      LFXM
01034:    000111  (0033)      CEA
01035:    000115  (0034)      LJIM
01036:    000117  (0035)      EJIM
01037:    000201  (0036)      IAB
01040:    000205  (0037)      PIM
01041:    000211  (0038)      PID
01042:    000215  (0039)      LPMJ
01043:    000217  (0040)      EPMJ
01044:    000217  (0041)      EVMJ
01045:    000311  (0042)      DIAG
01047:    000401  (0044)      ENB
01050:    000405  (0045)      OTK
01051:    000041  (0046)      CAI
01052:    000415  (0047)      ESIM
01053:    000417  (0048)      EVIM
01054:    000501  (0049)      LMCM
01055:    000503  (0050)      EMCM
01056:    000505  (0051)      SVC
01057:    000511  (0052)      ISI
01060:    000515  (0053)      OSI
01061:    001001  (0054)      INH
01062:    001011  (0055)      E64R
01063:    001013  (0056)      E32R
```

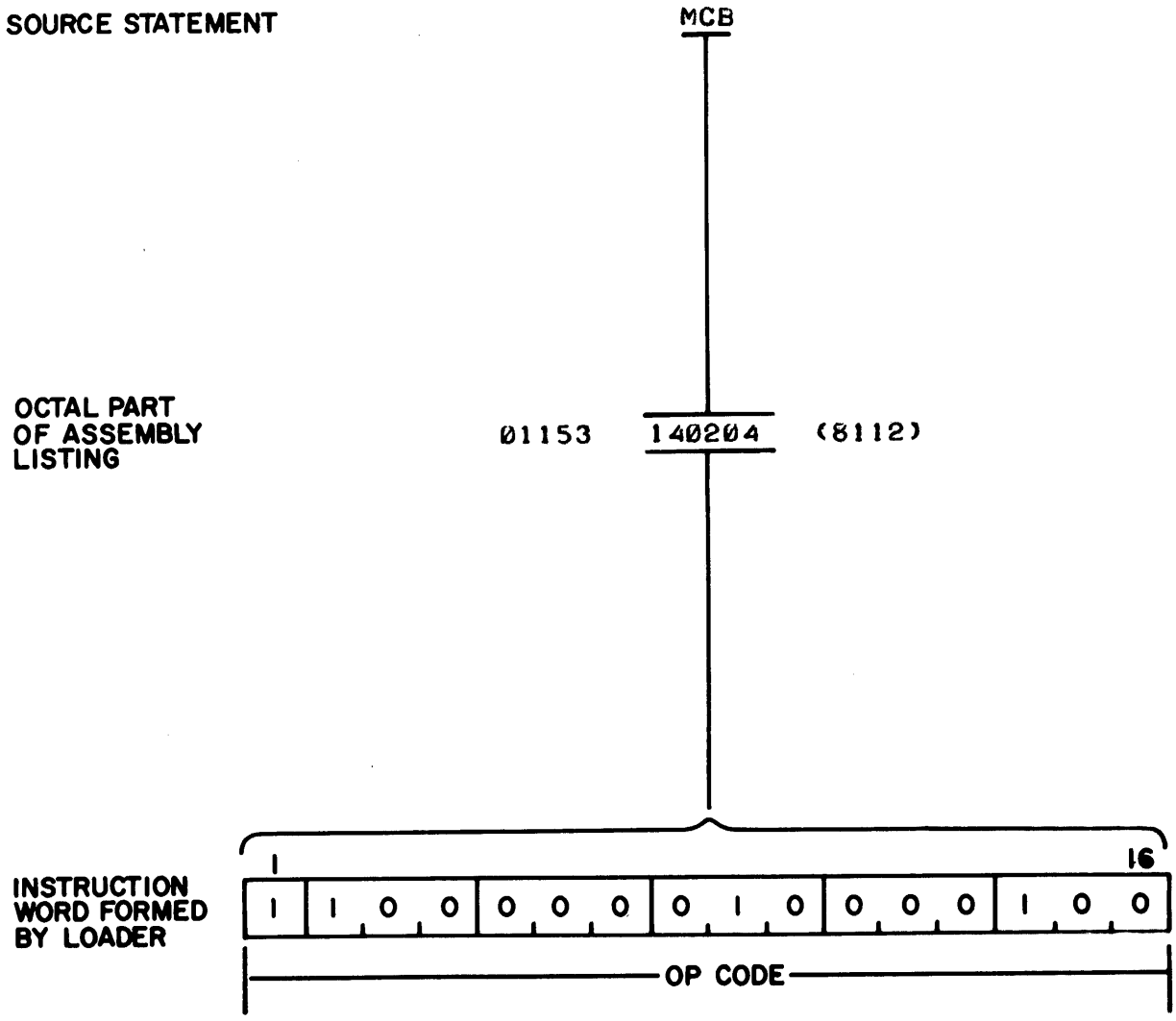


Figure 3-6. Assembly and Loading of Generic Instructions

```
01066:   001400 (0059)       ERM
01067:   100000 (0060)       SKP
01070:   101400 (0061)       SLT
01071:   101400 (0062)       SNI
01072:   100400 (0063)       SGE
01073:   100400 (0064)       SPL
01074:   101220 (0065)       SLE
01075:   100220 (0066)       SGT
01076:   100040 (0067)       SEQ
01077:   100040 (0068)       SZE
01100:   101040 (0069)       SNE
01101    101040 (0070)       SNZ
01102:   101001 (0071)       SSC
01103:   100001 (0072)       SRC
01104:   101200 (0073)       SSMC
01105:   101200 (0074)       SPS
01106:   100200 (0075)       SRMC
01107:   100200 (0076)       SPN
01110:   100100 (0077)       SLZ
01111:   101100 (0078)       SLN
01112:   101020 (0079)       SS1
01113:   100020 (0080)       SR1
01114:   101010 (0081)       SS2
01115:   100010 (0082)       SR2
01116:   101004 (0083)       SS3
01117:   100004 (0084)       SR3
01120:   101002 (0085)       SS4
01121:   100002 (0086)       SR4
01122:   101006 (0087)       SSS
01123:   100006 (0088)       SSR
01124:   140040 (0089)       CRA
01125:   140024 (0090)       CHS
01126:   140100 (0091)       SSP
01127:   140200 (0092)       RCB
01130:   140320 (0093)       CSA
01131:   140401 (0094)       CMA
01132:   140407 (0095)       TCA
01133:   140500 (0096)       SSM
01134:   140600 (0097)       SCB
01135:   141044 (0098)       CAR
01136:   141050 (0099)       CAL
01137:   141140 (0100)       ICL
01140:   141240 (0101)       ICR
01141:   141206 (0102)       AOA
01142:   141206 (0103)       A1A
01143:   141216 (0104)       ACA
01144:   141340 (0105)       ICA
01145:   140010 (0106)       CRL
01146:   140014 (0107)       CRB
01147:   140104 (0108)       XCA
01150:   140110 (0109)       SOA
01151:   140110 (0110)       S1A
01152:   140114 (0111)       IRX
01153:   140204 (0112)       XCB
01154:   140210 (0113)       DRX
01155:   140214 (0114)       CAZ
```

```
01156·    140304 (0115)       A2A
01157·    140310 (0116)       S2A
01160·    140410 (0117)       LLT
01161·    140411 (0118)       LLE
01162·    140412 (0119)       LNE
01163·    140413 (0120)       LEQ
01164·    140414 (0121)       LGE
01165·    140415 (0122)       LGT
```

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.)

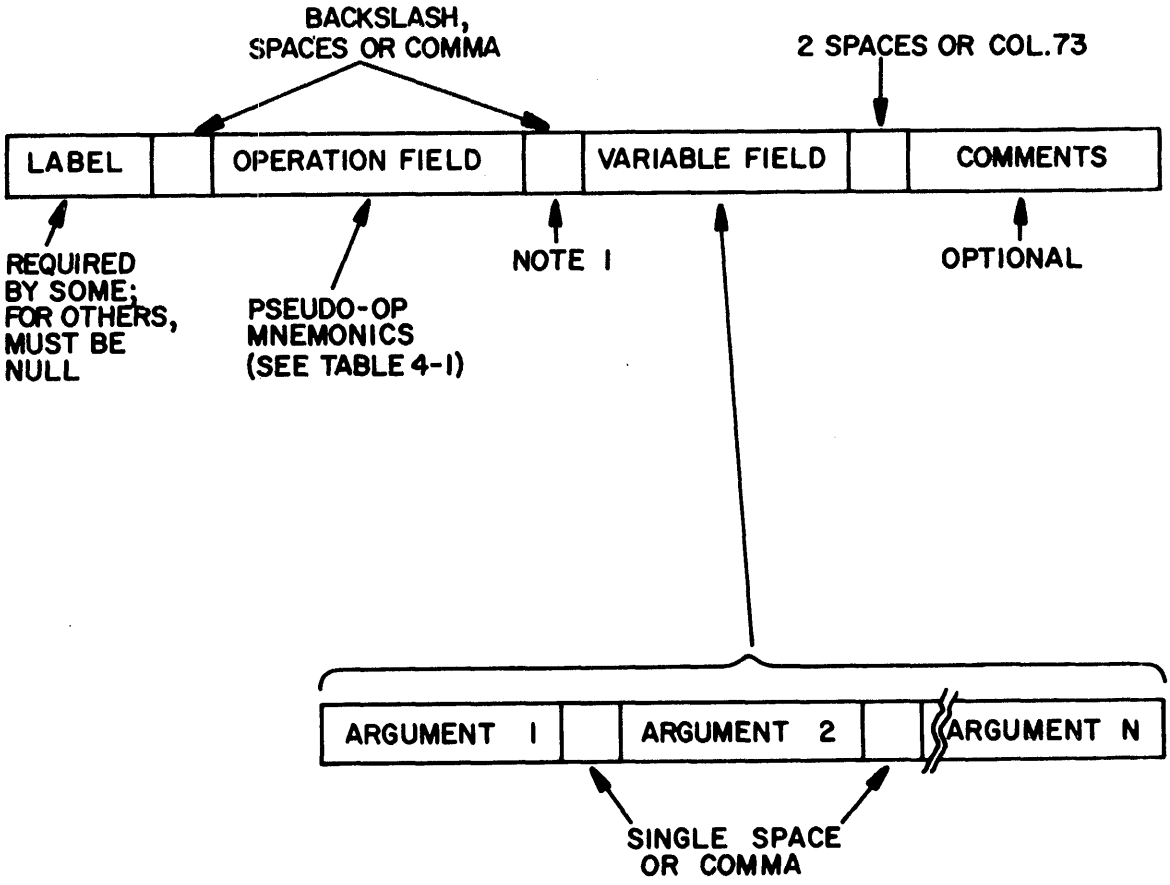
Symbolic names or other variables used in the variable field must be previously defined, unless otherwise stated in the pseudo-operation definition.

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.)

Table 4-1. Summary of Pseudo-Operations

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

* CLASSES:
AS - Assembly Control
CO - Conditional Assembly
DA - Data Defining
LI - Listing Control
LO - Loader Control
MA - Macro Definition
ST - Storage Allocation
SY - Symbol Defining



NOTES

1. IF MORE THAN 10 SPACES FOLLOW THE OPERATION FIELD, THE ASSEMBLER ASSUMES THERE IS NO VARIABLE FIELD AND TREATS THE REST OF THE LINE AS COMMENTS.

Figure 4-1. General Format of Pseudo-Operation Statements

ASSEMBLY CONTROLLING PSEUDO OPERATIONS

ABS (Set Mode to Absolute)

Label	Operation Field	Variable Field
Optional	ABS	Must be vacant

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)

Label	Operation Field	Variable Field
Not Used	REL	Must be vacant

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

ORG (Define Origin Location)

Label	Operation Field	Variable Field
Optional	ORG	Address Expression

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
                    (0002)          REL                 START RELOCATABLE
00000:    000001    (0003)          NOP
00001:    000001    (0004)          NOP
00002:  04 00607    (0005)          STA     SAVA        SAVE REGISTERS
00003:  15 00610    (0006)          STX     SAVX
00004:  12 00611    (0007)          IRS     CTR         UPDATE COUNTER
00005:  01 00600    (0008)          JMP     ABSO        JUMP TO ABSOLUTE LOCATION
          000600    (0009)          ORG     ^600
                    (0010)          ABS
00600:  02 00612    (0011) ABSO     LDA     =1          STARTS AT LOCATION ^600
00601:  04 00603    (0012)          STA     *+2
00602:  01 00604    (0013)          JMP     Y           RETURN TO RELOCATABLE
00603:    000000    (0015)          REL
00604:  02 00607    (0016) Y        LDA     SAVA        RESTORE REGISTERS
00605:  02 00610    (0017)          LDA     SAVX
00606:    000000    (0018)          HLT
00607:    000000    (0019) SAVA     DATA    0
00610:    000000    (0020) SAVX     DATA    0
00611:    000000    (0021) CTR      DATA    0
          000612    (0022)          END

00612:    000001
```

FIN (Insert Literals)

Label	Operation Field	Variable Field
Optional	FIN	Not used

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)

Label	Operation Field	Variable Field
Optional	MOR	Not used

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)

Label	Operation Field	Variable Field
Optional	END	Address expression

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)

Label	Operation Field	Variable Field
Not used	GO or GO TO	Statement label

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)

Label	Operation Field	Variable Field
Not used	LIST	Not used

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)

Label	Operation Field	Variable Field
Not used	NLST	Not used

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)

Label	Operation Field	Variable Field
Not used	EJCT	Not used

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)

Label	Operation Field	Variable Field
Optional	EXD	Not used

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)

Label	Operation Field	Variable Field
Optional	LXD	Not used

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)

Label	Operation Field	Variable Field
Optional	SETB	Address expression

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    *
05025:           (0006)  BSS     20       ALLOCATE 20 LOCATIONS FOR ADDRESS
                                          LINKS STARTING AT ^5001
```

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:

Label	Operation Field	Variable Field
Optional	DATA	One or more expressions, ASCII strings, or numerical data constants

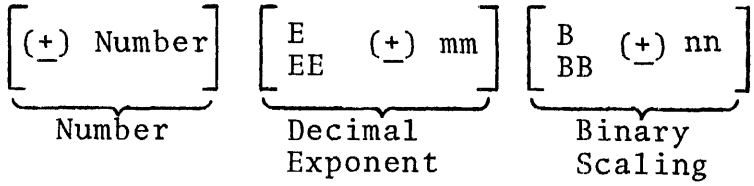
The current location is set equal to the expression(s) in the variable field. The variable field may contain any number of subfields, separated by commas. Subfields are assembled in consecutive locations starting with the leftmost subfield. If an expression requires more than one location (e.g. floating 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    C'A'
05027:   140702 (0009)          DATA    C'ABCDEF'
05030:   141704
05031:   142706
05032:   140702 (0010)          DATA    C'AB!'12'
05033:   123661
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:



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:

Hexadecimal	Octal	Decimal	Expressions
X'12AB'	0'1234'	12	X*2+3
X'-12AB'	0'-1234'	-12	ALPHA
$12AB	0'1234	0	Y .AND. '77
$-12AB	'1234		
X'12AB	'-1234		
$EFFF	'077777	32767	
$8000	'100000	-32768	
		1.23B6 (using binary scaling)	
		1.23E3B12 (using decimal exponent and binary scaling)	

Table 4-2. Numerical Formats in DATA Statements

Form of Number	Decimal Exponent (E or EE±mm)	Binary Scaling (B or BB±nn	Assembler Interprets Constant As:
Expression using Symbolic Variables	--	--	Single Precision Fixed Point
Hexadecimal	--	--	
Octal	--	--	
Decimal Integer	--	--	
Decimal Integer or Fraction	--	B	
	E	B	
	--	BB	Double Precision Fixed Point
	EE	BB	
	EE	--	Double Precision Floating Point
	E	--	Single Precision Floating Point
Decimal Fraction	--	--	

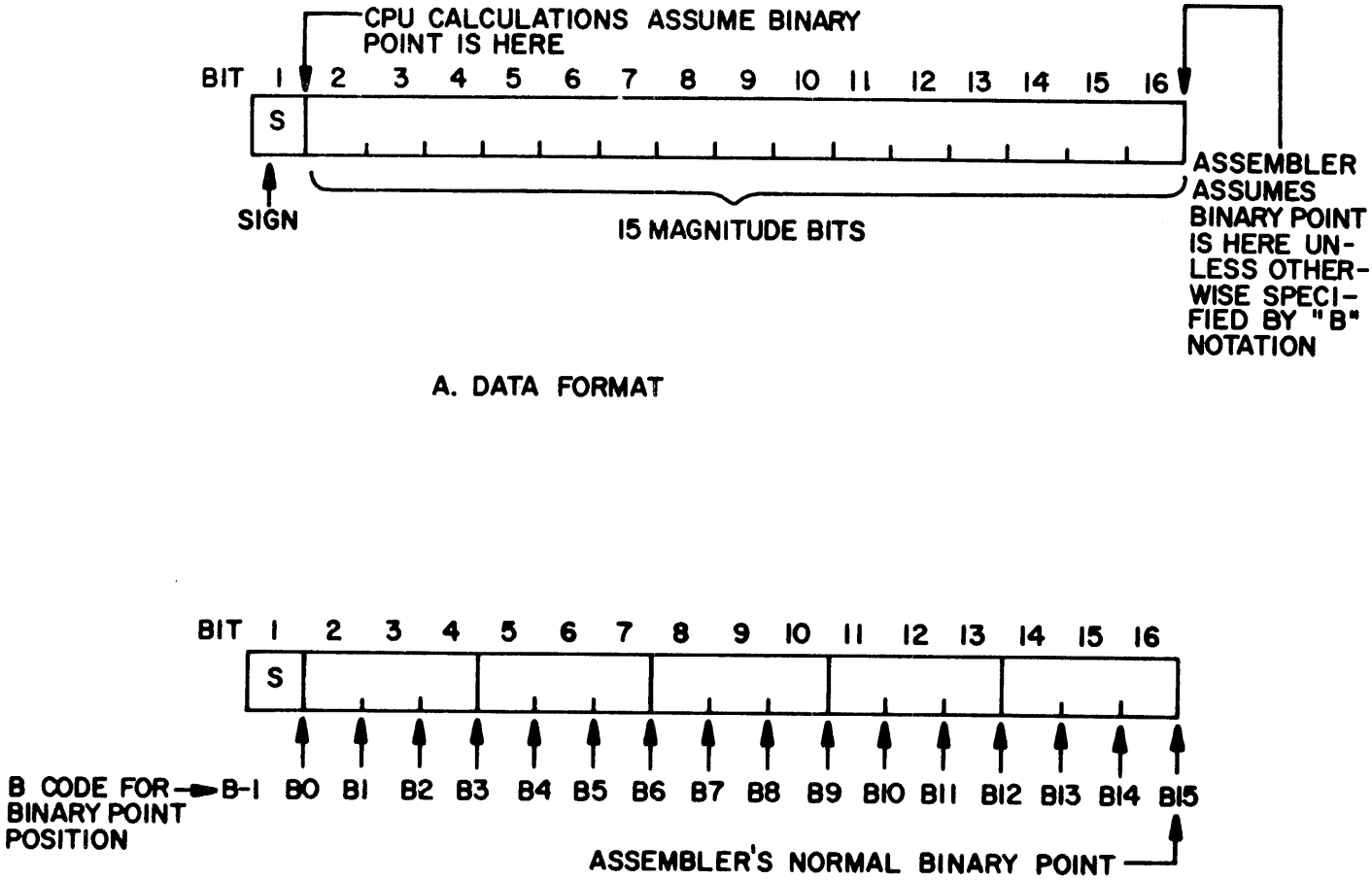


Figure 4-2. Single Precision Fixed Point Constants

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:

	Assembled As:	Decimal Equivalent
12.5B6	0 001 100^100 000 000 B6	6400
0.5B8	0 000 000 001^000 000 B8	64
5B8	0 000 001 010^000 000 B8	640

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 $K_{10}EmBn$ 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.

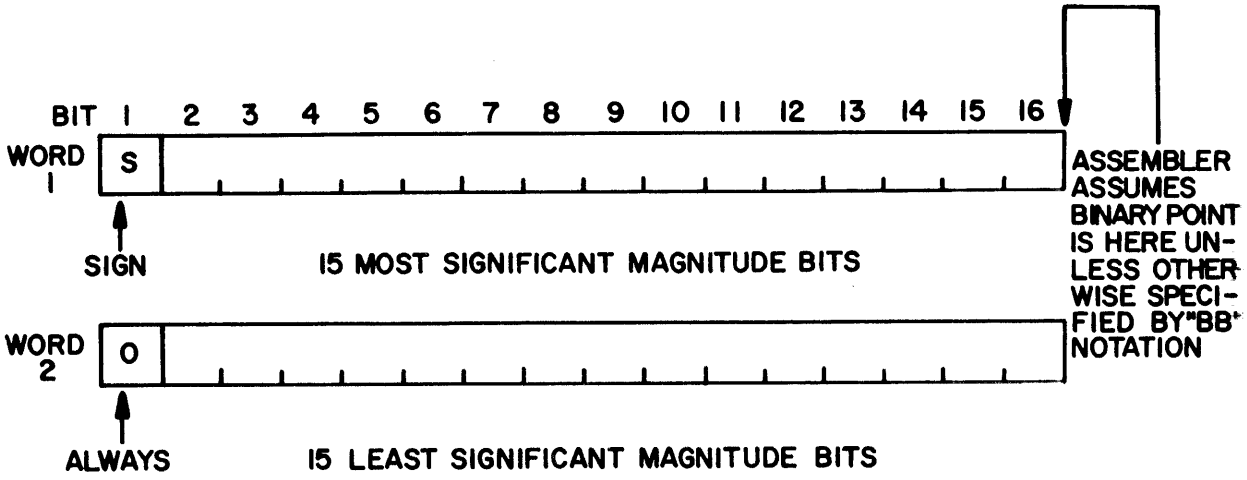
When expressed in DATA statements, fixed-point double precision constants must include a binary scale factor (BBn). A decimal exponent (EEn) is optional.

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.

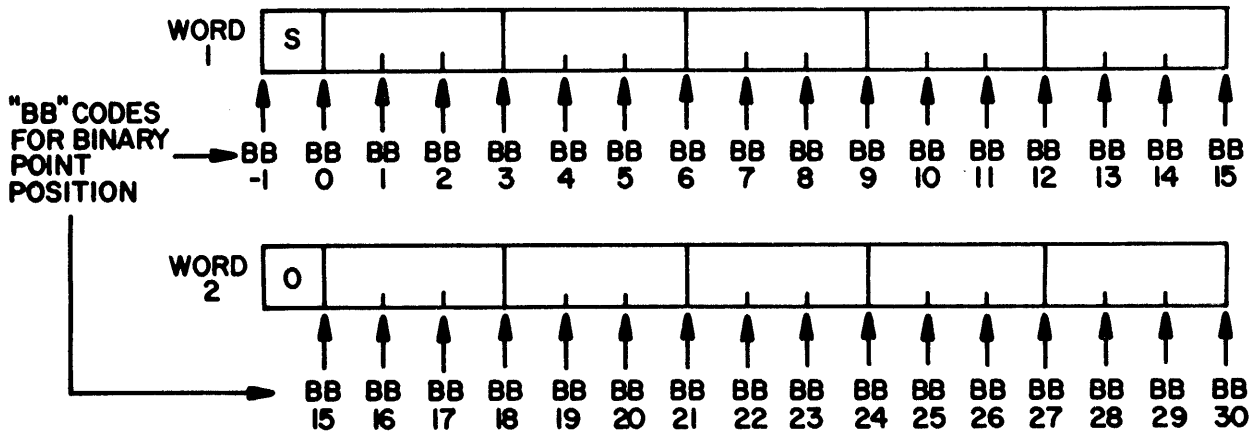
Examples:

		Assembled as:	Decimal Equivalent
12.5BB6	word 1	0 001 100 100 000 000	6400.00000
or 6.4EE3BB15	word 2	0 000 000 000 000 000	
7BB16	word 1	0 000 000 000 000 011	3.50000
	word 2	0 100 000 000 000 000	

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 (Emm). (Binary scaling must not be specified.)

Examples:

```
05046:   042100 (0017)          DATA   1.28E2
05047:   000000
05050:   040321 (0018)          DATA   1.28
05051:   165605
05052:   137456 (0019)          DATA   -1.28
05053:   012172
05054:   024563 (0020)          DATA   1.28E-14
05055:   045312
```

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 ($\pm$ 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 ($\pm$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.

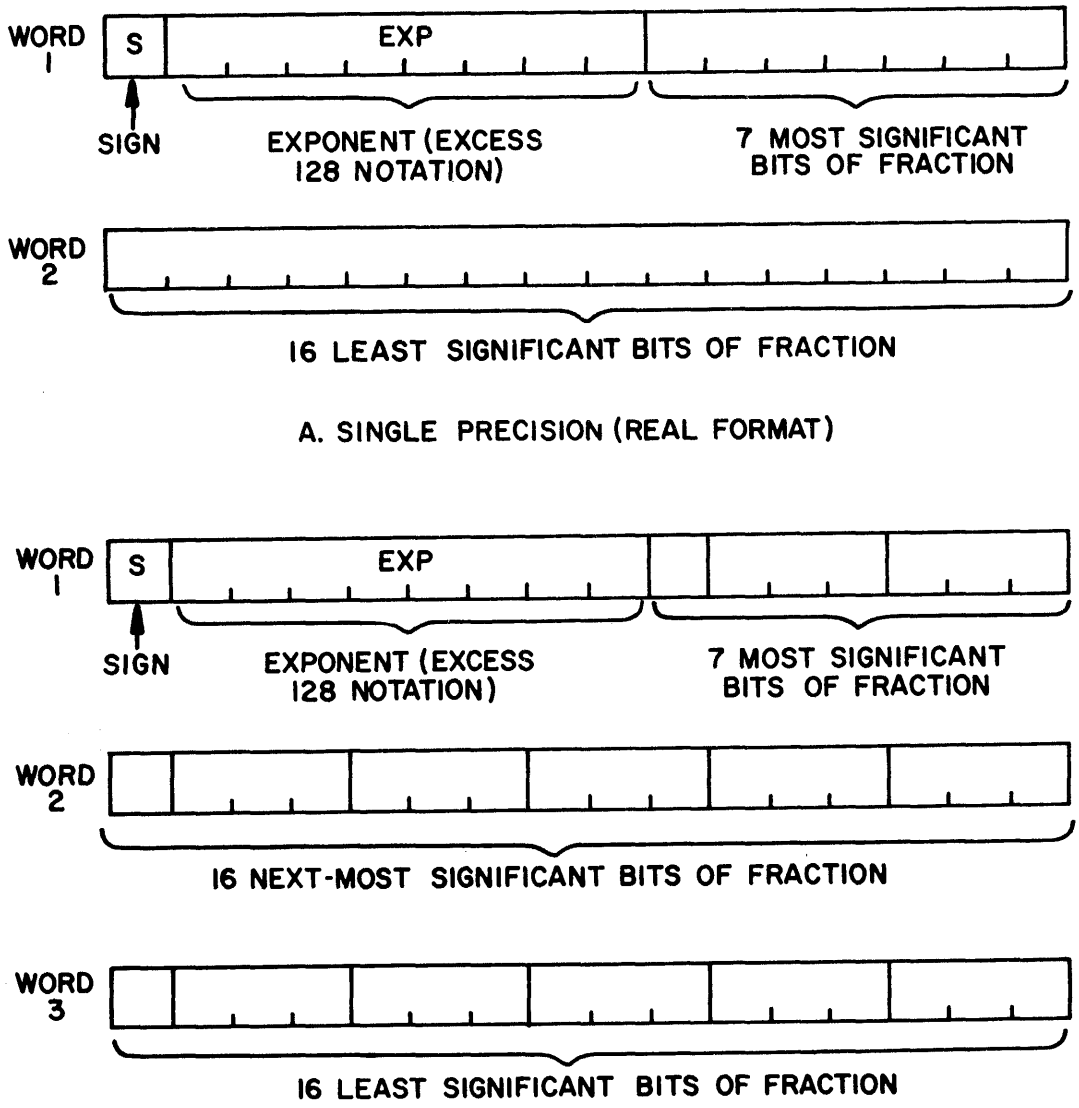


Figure 4-4. Floating Point Word Formats

Examples:

```
05056:    042100 (0021)          DATA    1.28EE2
05057:    000000
05060:    000000
05061:    135700 (0022)          DATA    -1.28EE2
05062:    000000
05063:    000000
05064:    024563 (0023)          DATA    1.28EE-14
05065:    045312
05066:    057537
```

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:    177776
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:    177776
05174:    177775
05175:    000000
05176:    000000
05177:    000000
05200:    000000
 . . .  .  .  .  .  .  .  .  .  .
05220:    000000
05221:    000012
05222:    005171
```

DEC (Set Decimal Constant)

Label	Operation Field	Variable Field
Optional	DEC	One or more decimal, octal, or hexadecimal constants (separated by commas)

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)

Label	Operation Field	Variable Field
Optional	DBP	One or more decimal, octal, or hexadecimal constants (separated by commas)

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:   000020
05316:   000000 (0044)          DBP   3,1.23B6,X'1A',$F,'123
05317:   000003
05320:   000000
05321:   040316
05322:   000000
05323:   000032
05324:   000000
05325:   000017
05326:   000000
05327:   000123
05330:   040316 (0045)          DBP   1.23                  FLOATING POINT
05331:   134121
05332:   127075 (0046)          DBP   -456.32E10,0.0,6.
05333:   114301
05334:   000000
05335:   000000
05336:   040740
05337:   000000
```

OCT (Set Octal Constant)

Label	Operation Field	Variable Field
Optional	OCT	One or more octal constants (separated by commas)

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:

```
05340:   000012 (0047)          OCT   12
05341:   070017 (0048)          OCT   70017,'321,677
05342:   000321
05343:   000677
05344:   177777 (0049)          OCT   '177777,-1,-'13,+177,+'177
05345:   177777
05346:   177765
05347:   000177
05350:   000177
```

HEX (Set Hexadecimal Constants)

Label	Operation Field	Variable Field
Optional	HEX	One or more hexadecimal constants (separated by commas)

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:

```
05351:    011253 (0050)          HEX    12AB
05352:    000017 (0051)          HEX    F, $F, -FFF1, -$FFF1, $-FFF1
05353:    000017
05354:    000017
05355:    000017
05356:    000017
05357:    011253 (0052)          HEX    $12AB, +12AB, +$12AB, $+12AB
05360:    011253
05361:    011253
05362:    011253
```

VFD (Define Variable Fields)

Label	Operation Field	Variable Field
Optional	VFD	One or more subfields of the form: Field Size,Value

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)

Label	Operation Field	Variable Field
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
```

DAC (Local Address Definition)

Label	Operation Field	Variable Field
Optional	DAC	Address Expression
	or	or
	DAC* (indirect addressing)	Address Expression,1 (Indexing)

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)

Label	Operation Field	Variable Field
Optional	XAC	External variable
	or	or
	XAC* (Indirect addressing)	External Variable,1 (Indexing)

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$1,1
05424: 40.00000 (0071)          XAC*    T$1
05425: 40.00000 (0072)          XAC*   T$1,1
```

*** (Dummy Memory Reference Instruction)

Label	Operation Field	Variable Field
Optional	***	Address Expression
	or	or
	**** (Indirect Addressing)	Address Expression,1 (Indexing)

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)

	Label	Operation Field	Variable Field
Format A	Contains a variable	EQU or SET	Address Expression
Format B	Blank	EQU or SET	One or more symbol equality expressions (separated by commas)

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=$39FF,K='7777,L=C'AZ'
000022 (0079)         SET    K=18
035376 (0080)         SET    K=J+$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:

```
                    (0001) *DEMONSTRATES EQU AND SET
                    (0002)              REL
            000020  (0003) CHAN1        EQU     '20             DMA CHANNEL 1
            000015  (0004) STRTADR      SET     BUF1            STARTING ADDRESS I/O TRANSFER
00000:      000001  (0005)              NOP
00001:      000001  (0006)              NOP
00002:      000001  (0007)              NOP
00003:   02.00015   (0008)              LDA     STRTADR
00004:   04.00020   (0009)              STA     CHAN1           SET STARTING ADDRESS TO BUF1
00005:      000001  (0010)              NOP
00006:      000001  (0011)              NOP
00007:      000001  (0012)              NOP
            000041  (0013)              SET     STRTADR=BUF2 CHANGE STARTING ADDRESS
00010:   02.00041   (0014)              LDA     STRTADR
00011:   04.00020   (0015)              STA     CHAN1           SET STARTING ADDRESS TO BUF2
00012:      000001  (0016)              NOP
00013:      000001  (0017)              NOP
00014:      000001  (0018)              NOP
00064:      000065  (0021)              END
```

```
BUF1        000015/ 0004   0019
BUF2        000041/ 0013   0020
CHAN1       000020  0003   0009   0015
STRTADR     000041  0004   0008   0013   0014
```

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

Label	Operation Field	Variable Field
Optional	BSS, BES or BSZ	Expression that specifies number of words to be allocated

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)

Label	Operation Field	Variable Field
Not used	SETC	Address expression

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)

Label	Operation Field	Variable Field
Optional	COMN	One or more variables (separated by commas)

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          SET COMMON BASE
  013777        (0004) COMN   AA,AB,AC        ASSIGN THREE LOCATIONS IN COMMON
  013776
  013775
00000: 02 00004 (0005) LDA    =1
00001: 04 13777 (0006) STA    AA              SET FLAG AA
00002: 12 13775 (0007) IRS    AC              UPDATE COUNTER AC
00003:   000000 (0008) HLT                    FINISHED
         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    BA,BB,BC,BD   ASSIGN FOUR LOCATIONS IN COMMON
                  013776
                  013775
                  013774
000000: 02 13777  (0006)  LDA     BA              CHECK PROGRAM FLAG
000001:   100040  (0007)  SZE
000002: 12 13774  (0008)  IRS     BD              UPDATE COUNTER BD
000003: 02 000006 (0009)  LDA     =1
000004: 04 13776  (0010)  STA     BB              SET FLAG BB
                  000006 (0012)  END

000006:   000001


        BA          013777/ 0005  0006
        BB          013776/ 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 (CB) 10,20,10
000000    JMP   000000
          LINK  000000
000001    CALL  I.$22
000002    DAC   CB        +
C  UPDATE COUNTER CE
000003    SNZ
000004    JMP   _20
10    CE=CE+1.
C  SET FLAG OF PROGRAM C
000005    CALL  L.$22
000006    DAC   CE        +
000007    CALL  A.$22
000010    DAC   ='040300
000011    CALL  H.$22
000012    DAC   CE        +
20    CF=1.
          LINK  _20
000013    LDA   ='000001
000014    CALL  C.$12
000015    CALL  H.$22
000016    DAC   CF        +
      STOP
000017    CALL  F.$HT
000020    DAC   ='000001
000021    JMP   000017
      END
000005    DAC   _10
000013    DFC   _20
          LINK  ='040300
000022    OCT   040300
000023    'OCT  000000
          LINK  ='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)

Label	Operation Field	Variable Field
Optional	EXT	One or more external entry point names

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)

Label	Operation Field	Variable Field
Optional	CALL	External Entry Point
	(* for indirect addressing is optional)	(,1 for indexing is optional)

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:

Label	Operation Field	Variable Field
Optional	SUBR or ENT	Extname or Extname, Entryname

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            SUBR  TST1
Subroutine            .
                      .
                      .
                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                CALL  MAINT1
Program              .
                     .
                     .

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

External            SUBR  MAINT1,TST1
Subroutine           .
                     .
                     .
              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                CALL  MAINT1,TST1
Program              .
                     .
                     .
                    LDA   TST2
                     .
                     .
                     .
                    JMP   TST3
                     .
                     .
                     .
                    EXT   TST2
                    EXT   TST3
-------------------------------------
```

```
External              SUBR  MAINT1,TST1
Subroutine            ENT   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                  JMP   TEST2
Program                .
                       .
                       .
                      EXT   TEST2
                       .
                       .
                       .

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

External
Subroutine             ENT   TEST2,TST2
                       .
                       .
                       .
                TST2  LDA   XYZ
                       .
                       .
                       .
```

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)

Label	Operation Field	Variable Field
Optional	IF	(Expression)(Statement) : (Statement): (etc.)

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.

Label	Operation Field	Variable Field
Optional	IFM IFP IFZ IFN	Expression

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 ≥ 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 ≠ 0)
```

ENDC (End Conditional Assembly Area)

Label	Operation Field	Variable Field
Not Used	END C	Not Used

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)

Label	Operation Field	Variable Field
Not Used	ELSE	Not Used

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)       IGP    MTWO
                    (0080)       IFP    FIVE
                    (0082)       ELSE
                    (0084)       ENDC
                    (0086)       ELSE
01002: 02.00007     (0087)       LDA    7
                    (0088)       ENDC
```

FAIL (force Error Message)

Label	Operation Field	Variable Field
Optional	FAIL	Not used

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)

Label Field	Operation Field	Variable Field
Name of macro (to be used in operation field of macro calls)	MAC	Optional dummy words and/or argument identifiers (see text) separated by commas.

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:

Label	Operation Field	Variable Field
Optional	ENDM	Ignored

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 <1> 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:

Label	Operation Field	Variable Field
Optional	Name of User-Defined or Library Macro	Argument value expressions, plus optional dummy words or argument identifiers, separated by commas or blanks

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:

Variable Field	Argument <1>	Argument <2>	Argument <3>
A	A	0	0
A+3	A+3	0	0
X,Y-1,Z*A-1	X	Y-1	Z*A-1
X,B-C (Z3X2)	X	B-C	Z3X2
(A,B-1), C	A,B-1	C	
(X,Y,(Z1+Z2),3)	X,Y,Z1+Z2),3	0	0

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                WAIT      MAC
DEFINITION                     IRS <1>
CONTAINING                     JMP * -1
CALL TO                        TRANSFER <2>
"TRANSFER"                     ENDM
MACRO

CALL TO
WAIT MACRO                     WAIT 100, (ARG1, ARG2)


ASSEMBLED                      IRS 100
AS                             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
              :
              :
              :
              :
            ENDM
```

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:

MAC Statement	Macro Call
TRANSFER MAC DATA, FROM, TO	TRANSFER DATA FROM ALPHA TO '7770
TRANSFER MAC VOLTS,TO,DIG OUT	TRANSFER 3.22 VOLTS TO DIG OUT 14
TRANSFER MAC COUNTS,TO,PULSER	TRANSFER 3374 COUNTS TO PULSER $FF

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>
```

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.

Examples:	Local Label	Evaluated As	In Macro Call
	&ABC	0002ABC	0002
	&X3A	1739X3A	1739

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)

Label	Operation Field	Variable Field
Optional	LSTM	Not used

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)           LSTM
                001000  (0004) START     ORG     '1000
                        (0005) TRANSFER  MAC     TO
                        (0006)           LDA     <1>
                        (0007)           OTA     <2>
                        (0008)           ENDM
01000: 02 01002 (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)

Label	Operation Field	Variable Field
Not used	LSMD	Not used only those

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)

Label	Operation Field	Variable Field
Not used	NLSM	Not used

Inhibits listing of statements generated by the assembler during macro expansion. Only the macro call is listed.

BACK, BACK TO (Loop Back - Macros Only)

Label	Operation Field	Variable Field
Optional	BACK or BACK TO	Statement label

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)

Label	Operation Field	Variable Field
Optional	SAY	Any ASCII text string

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) OLDMAC 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 <1>, 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)  OLDMAC, 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)  OLDMAC, 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)  OLDMAC, 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] [BEEP],
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
```

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> .EQ. 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, @ DECIMAL
 POSITIONS

 CODE FIELD OF INPUTREC FROM COLUMNS 25 TO 30

 NAME FIELD OF INPUTREC FROM COLUMNS 65 TO 75

 NAMEOUT FIRLF 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

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

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

Label	Operation Field	Variable Field
Optional	$COPY	m,n

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

$DONE

Label	Operation Field	Variable Field
Optional	$DONE	Not used

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

APPENDICES

APPENDIX A

PRIME 200 INSTRUCTIONS

(OP CODE ORDER)

G	000000		HLT	HALT
G	000001		NOP	NO OPERATION
G	000005		SGL	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		EMCM	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
IO      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 (S1A) 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
```

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 CLEARS 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)

BR	100240+N	SRN	SKIP ON SENSE SWITCH N RESET
BR	100260+N	SARN	SKIP ON A BIT N RESET
BR	101240+N	SSN	SKIP ON SENSE SWITCH N SET
BR	101260+N	SASN	SKIP ON A BIT N SET
G	000000	HLT	HALT
G	000001	NOP	NO OPERATION
G	000005	SGL	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	EMCM	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
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
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
G	101400	SMI	(SLT)	SKIP ON A MINUS
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	(S1A)	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
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
IO	14	OCP		OUTPUT CONTROL PULSE
IO	34	SKS		SKIP IF SET
IO	54	INA		INPUT TO A
IO	74	OTA		OUTPUT FROM A
IO	74	SMK		SET INTERRUPT MASK
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
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

MR	13	IMA		INTERCHANGE MEMORY AND A
MR	15	LDX		LOAD INDEX (ASSEMBLER SETS INDEX BIT)
MR	15	STX		STORE INDEX (ASSEMBLER CLEARS INDEX BIT)
MR	16	MPY		MULTIPLY
MR	17	DIV		DIVIDE
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

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	(A1A)	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	CAI		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	CAZ		COMPARE A WITH ZERO
G	000111	CEA		COMPUTE EFFECTIVE ADDRESS
G	140024	CHS		CHANGE SIGN OF A
G	140401	CMA		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 *	DSB		DOUBLE PRECISION SUBTRACT
MR	04 *	DST		DOUBLE PRECISION STORE
G	000011	E16S	(EXA)	ENTER 16K SECTOR ADDRESSING MODE
G	001013	E32R		ENTER 32K RELATIVE ADDRESSING MODE
G	000013	E32S	(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	ESIM		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
G	141240	ICR		INTERCHANGE BYTES OF A AND CLEAR RIGHT BYTE

MR	13	IMA	INTERCHANGE MEMORY AND A
IO	54	INA	INPUT TO A
G	001001	INH	INHIBIT INTERRUPT
G	000043	INK	TRANSFER (INPUT) STATUS KEYS TO A
MR	12	IRS	INCREMENT, REPLACE MEMORY AND SKIP
G	140114	IRX	INCREMENT, REPLACE INDEX AND SKIP
G	000511	ISI	INPUT SERIAL INTERFACE TO A
MR	01	JMP	UNCONDITIONAL JUMP
MR	10	JST	JUMP TO EA + 1 AND STORE P IN EA
MR	02	LDA	LOAD A
MR	15	LDX	LOAD INDEX (ASSEMBLER SETS INDEX BIT)
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	140411	LLE	CONVERT A<=0 TO TRUE
SH	0410NN	LLL	LONG LEFT LOGICAL
SH	0412NN	LLR	LONG LEFT ROTATE
SH	0411NN	LLS	LONG LEFT SHIFT
G	140410	LLT	CONVERT A<0 TO TRUE
G	000501	LMCM	LEAVE MACHINE CHECK MODE
G	140412	LNE	CONVERT ^(A=0) TO TRUE
SH	0400NN	LRL	LONG RIGHT LOGICAL
SH	0402NN	LRR	LONG RIGHT ROTATE
SH	0401NN	LRS	LONG RIGHT SHIFT
MR	16	MPY	MULTIPLY
G	000001	NOP	NO OPERATION
G	000101	NRM	NORMALIZE
IO	14	OCP	OUTPUT CONTROL PULSE
G	000515	OSI	OUTPUT SERIAL INTERFACE FROM A
IO	74	OTA	OUTPUT FROM A
G	000405	OTK	TRANSFER (OUTPUT) A TO STATUS KEYS
G	000211	PID	POSITION FOR INTEGER DIVIDE
G	000205	PIM	POSITION FOR INTEGER MULTIPLY
G	140200	RCB	RESET C-BIT
G	000021	RMC (RMP)	RESET MACHINE CHECK
G	140310	S2A	SUBTRACT TWO FROM A
BR	100260+N	SARN	SKIP ON A BIT N RESET
BR	101260+N	SASN	SKIP ON A BIT N SET
G	000041	SCA	TRANSFER SHIFT COUNTER TO A
G	140600	SCB	SET C-BIT
G	000005	SGL	ENTER SINGLE PRECISION MODE
G	100220	SGT	SKIP ON A GREATER THAN ZERO
G	100000	SKP	UNCONDITIONAL SKIP
IO	34	SKS	SKIP IF SET
G	101220	SLE	SKIP ON A LESS THAN OR EQUAL TO ZERO
G	101100	SLN	SKIP ON A BIT 16 ONE
G	100100	SLZ	SKIP ON A BIT 16 ZERO
G	100200	SMCR(SPN)	SKIP ON MACHINE CHECK RESET
G	101200	SMCS(SPS)	SKIP ON MACHINE CHECK SET
G	101400	SMI (SLT)	SKIP ON A MINUS
IO	74	SMK	SET INTERRUPT MASK
G	101040	SNZ (SNE)	SKIP ON A NOT ZERO
G	140110	SOA (S1A)	SUBTRACT ONE FROM A
G	100400	SPL (SGE)	SKIP ON A PLUS
G	100001	SRC	SKIP ON C-BIT RESET
BR	100240+N	SRN	SKIP ON SENSE SWITCH N RESET

G	101001	SSC		SKIP ON C-BIT SET
G	140500	SSM		SET SIGN OF A MINUS
BR	101240+N	SSN		SKIP ON SENSE SWITCH N SET
G	140100	SSP		SET SIGN OF A PLUS
G	100036	SSR		SKIP ON NONE OF SENSE SWITCHES 1-4 SET
G	101036	SSS		SKIP ON ANY OF SENSE SWITCHES 1-4 SET
MR	04	STA		STORE A
MR	15	STX		STORE INDEX (ASSEMBLER CLEARS INDEX BIT)
MR	07	SUB		SUBTRACT MEMORY FROM A
G	000505	SVC		SUPERVISOR CALL
G	100040	SZE	(SEQ)	SKIP ON A ZERO
G	140407	TCA		TWO'S COMPLEMENT A
G	000311**	VIRY		VERIFY
G	140104	XCA		TRANSFER A TO B AND CLEAR A
G	140204	XCB		TRANSFER B TO A AND CLEAR B

APPENDIX D

I/O DEVICE CODES

APPENDIX E

ASCII CHARACTER CODES

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

APPENDIX F

OBJECT FILE FORMATS

APPENDIX G

ASSEMBLER ERROR MESSAGES

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

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 X 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.

PMA LOADER CONTROL PSEUDO-OPS

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:

0	16K Sectored Mode
1	32K Sectored Mode
2	64K Relative Mode
3	32K Relative Mode

The SDM pseudo-op does not change the current desectorization mode.

DDM - Desector 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 SDM pseudo-op.

REV. 4 PMA EXTENSIONS

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:

   ```
   B'101' = 5
   %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    exp1
```

where:

exp1 - bit mask defining instruction groups that are to trigger loading. Bit assignments are the same as for DUII.

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 statment 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.

+	mode of right operand: abs	m*rel	stack + j*rel
mode of left operand: abs	abs	m*rel	stack + j*rel
n*rel	n*rel	(m+n)*rel	stack + (j+n)*rel
stack + i*rel	stack + i*rel	stack + (i+m)*rel	P

mode of left operand	mode of right operand: -	abs	m*rel	stack + j*rel
	abs	abs	-m*rel	P
	n*rel	n*rel	(n-m)*rel	P
	stack + i*rel	stack + i*rel	stack + (i-m)*rel	P

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 consistant 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 $<$ '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.

```
                        (0001) *        EXAMPLES OF REV.  4 PMA EXTENSIONS
                        (0002) *
                        (0003) *
                        (0004) *
                        (0005) *
                        (0006)          REL
                        (0007) *
                        (0008) *
                        (0009) *                ************************
                        (0010) *                * NEW CONSTANT FORMS *
                        (0011) *                ************************
                        (0012) *
                        (0013) *
                        (0014) *
000000:   000161        (0015)          DATA    B'1110001'      BINARY CONSTANT
000001:   000161        (0016)          DATA    %1110001        SAME CONSTANT, DIFFERENT FORM
000002:   000303        (0017)          DATA    R'C',R'/'       SINGLE CHARACTER CONSTANTS
000003:   000247
                        (0018) *
                        (0019) *
                        (0020) *
                        (0021) *                ********************
                        (0022) *                * C64R PSEUDO-OP *
                        (0023) *                ********************
                        (0024) *
                        (0025) *
                        (0026)          C64R
S 000004: 40.000000A    (0027)          DAC*    0               FLAG INDIRECT DAC
S 000005: 42.000406     (0028)          LDA*    *+257           INDIRECT, OUT OF RELATIVE REACH
                        (0029) *
                        (0030) *
                        (0031) *                ********************
                        (0032) *                * SETB PSEUDO-OP *
                        (0033) *                ********************
                        (0034) *
                        (0035) *
          000031        (0036)          SETB    BASE,25
                        (0037) *
```

```
000006:                (0038) BASE    BSS     25
                       (0039) *
                       (0040) *
                       (0041) *
                       (0042) *               **********************
                       (0043) *               * DUII PSEUDO-OP *
                       (0044) *               **********************
                       (0045) *
                       (0046) *
                       (0047) *
                       (0048)         DUII    '14,1           DUII FOR A UII PACKAGE THAT SATISFIES
                       (0049) *                               SINGLE AND DOUBLE PRECISION FLOATING POINT AND
                       (0050) *                               REQUIRES HIGH SPEED ARITHMETIC FOR EXECUTION
                       (0051) *
                       (0052) *
                       (0053) *               ********************
                       (0054) *               * LIR PSEUDO-OP *
                       (0055) *               ********************
                       (0056) *
                       (0057) *
                       (0058) *
                       (0059)         LIR     %1100           LOAD IF SINGLE AND/OR DOUBLE FLOATING POINT USED
                       (0060) *
                       (0061) *
                       (0062) *
                       (0063) *               **********************
                       (0064) *               * DYNM PSEUDO-OP *
                       (0065) *               **********************
                       (0066) *
                       (0067) *
          000002       (0068)         DYNM    ADDRESS,NAME(3),BUFFER(80),TEMP1,TEMP2
          000003
          000006
          000126
          000127
000037:   003414       (0069)         ENTR    #107
000040: 00.000130A
                       (0070) *
```

```
                000003    (0071)          DYNM    ABC=3,DEF(4)=5
                000005
                000024    (0072)          DYNM    =20,T1,T2,T3
                000024
                000025
                000026
                          (0073) *
                          (0074) *
                          (0075) *
                          (0076) *                ****************************************
                          (0077) *                * GENERATION OF ALL FORMS OF A LDA *
                          (0078) *                ****************************************
                          (0079) *
000041:                   (0080) LOCAL   BSS      1
                          (0081) *
                          (0082) *        FIRST, THE FOUR NON-SPECIAL RELATIVE FORMS
                          (0083) *
000042: 02.000041         (0084)          LDA     LOCAL
000043: 22.000041         (0085)          LDA     LOCAL,1
000044: 42.000041         (0086)          LDA*    LOCAL
000045: 62.000041         (0087)          LDA*    LOCAL,1
                          (0088) *
                          (0089) *        THE 16 SPECIAL RELATIVE FORMS
                          (0090) *
000046:   005400          (0091)          LDA%    LOCAL           '%' REQUIRED BECAUSE SHORT FORM AVAILABLE
000047: 00.000041
000050:   005401          (0092)          LDA     @+3
000051: 00.000035
000052:   005402          (0093)          LDA     @+
000053:   005403          (0094)          LDA     -@
                          (0095) *
000054:   045400          (0096)          LDA%    LOCAL,1
000055: 00.000041
000056:   045401          (0097)          LDA     NAME+3,1        (NAME WAS DECLARED IN A DYNM)
000057: 00.000065
000060:   045402          (0098)          LDA     @+,*1
000061:   045403          (0099)          LDA     -@,*1
                          (0100) *
```

```
000062:    105400    (0101)          LDA%    LOCAL,*
000063:  00.000041
000064:    105401    (0102)          LDA     TEMP1,*
000065:  00.0001265
000066:    105402    (0103)          LDA*    @+
000067:    105403    (0104)          LDA     --@,*
                     (0105) *
000070:    145400    (0106)          LDA     LOCAL,1*        PREINDEXED, >= '100
000071:  00.000041
000072:    145401    (0107)          LDA     ADDRESS,1*
000073:  00.0000025
000074:    145402    (0108)          LDA     '30,*1          POST-INDEXED, <'100
000075:  00.000030A
000076:    145403    (0109)          LDA     ADDRESS,*1
000077:  00.0000025
                     (0110) *
                     (0111) *
                     (0112) *        ****************************************
                     (0113) *        * LEGAL + ILLEGAL EXPRESSION MODES *
                     (0114) *        ****************************************
                     (0115) *
                     (0116) *
                     (0117) *
  000100:  02.000100 (0118)          LDA     *+++++*--*-*-*  RESULT IS 1*REL, OK
Z 000101:  02.000202A (0119)         LDA     *++++-*         RESULT IS 2*REL, ERROR
  000102:    005401  (0120)          LDA     @+*-*           RESULT IS STACK + ABS, OK
  000103:  00.0000005
Z 000104:  02.000104A (0121)         LDA     @+*             RESULT IS STACK + REL, ERROR
Z 000105:  02.000105A (0122)         LDA     *-@             CANNOT SUBTRACT STACK RELATIVE
Z 000106:  02.000000A (0123)         LDA     @+@             CANNOT ADD 2 STACK RELATIVES
                     (0124) *
                     (0125) *
                     (0126) *
            000107   (0127)          END
```

```
ABC         0000035S 0071
ADDRESS     0000025S 0068  0107  0109
BASE        000006   0036  0038
BUFFER      0000065S 0068
DEF         0000055S 0071
LOCAL.      000041   0000  0004  0005  0006  0007  0091  0096  0101  0106
NAME        0000035S 0068  0097
T1          0000245S 0072
T2          0000255S 0072
T3          0000265S 0072
TEMP1       0001265S 0068  0102
TEMP2       0001275S 0068

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:

```
Label      COMM          S1, S2,....Sn
```

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:

Counter	Initial Value
sc	0
sm	0

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

PRIME

PRIME Computer, Inc., 145 Pennsylvania Avenue, Framingham, Massachusetts 01701