Xerox Symbol

Sigma 5-9 Computers

Language and Operations Reference Manual

90 17 90A

SYMBOL DIRECTIVES

			Page No.
[label]	ASECT		19
	BOUND	boundary	16
name	COM, field list	value list	27
[label]	CSECT	[value]	19
[label]	DATA[,f]	$value_1$[,...,$value_n$]	28
	DEF	$symbol_1$[,...,$symbol_n$]	25
[label]	DO1	exp	22
	END	[exp]	22
label	EQU	exp	24
[label]	GEN, field list	value list	26
	GOTO[,k]	$label_1$[,...,$label_n$]	22
[label]	LOC[,n]	location	16
	LOCAL	[$name_1$,...,$name_n$]	23
[label]	ORG[,n]	location	16
	PAGE		30
	REF	$symbol_1$[,...,$symbol_n$]	25
[label]	RES[,n]	u	18
	SREF	$symbol_1$[,...,$symbol_n$]	25
	SYSTEM	name	21
[label]	TEXT	'cs'	30
[label]	TEXTC	'cs'	30

XEROX

Xerox Symbol

Sigma 5-9 Computers

Language and Operations

Reference Manual

FIRST EDITION

90 17 90A

June 1971

Printed in U.S.A.

NOTICE

This publication, 90 17 90A, documents version H00 of Xerox Sigma 5-9 Symbol.

RELATED PUBLICATIONS

Title	Publication No.
Xerox Sigma 5 Computer/Reference Manual	90 09 59
Xerox Sigma 6 Computer/Reference Manual	90 17 13
Xerox Sigma 7 Computer/Reference Manual	90 09 50
Xerox Sigma 8 Computer/Reference Manual	90 17 49
Xerox Sigma 9 Computer/Reference Manual	90 17 33
Xerox Basic Control Monitor (BCM)/BP, RT Reference Manual	90 09 53
Xerox Batch Processing Monitor (BPM)/BP, RT Reference Manual	90 09 54
Xerox Batch Time-Sharing Monitor (BTM)/TS Reference Manual	90 15 77
Xerox Batch Time-Sharing Monitor (BTM)/TS User's Guide	90 16 79

Manual Type Codes: BP - batch processing, LN - language, OPS - operations, RBP - remote batch processing, RT - real-time, SM - system management, TS - time-sharing, UT - utilities.

The specifications of the software system described in this publication are subject to change without notice. The availability or performance of some features may depend on a specific configuration of equipment such as additional tape units or larger memory. Customers should consult their XDS sales representative for details.

CONTENTS

APPENDIXES

FIGURES

TABLES

EXAMPLES

1. INTRODUCTION

Xerox Symbol is a one-pass assembler that reads source language programs and converts them to object language programs. Symbol outputs an object program and an assembly listing. The object language format is explained in the BCM BP, RT Reference Manual, 90 09 53; the format of the assembly listing is described in Chapter 6 of this manual.

PROGRAMMING FEATURES

Symbol provides such programming features as forward references, literals, and external definitions. Since these types of items cannot be defined by a single-pass assembler, Symbol produces information that enables the loader to provide the appropriate linkages at load time.

Other features of Symbol:

- Self-defining constants that facilitate use of hexadecimal, decimal, octal, floating-point, and fixed-point values.
- The facility for writing large programs in segments or modules. The assembler provides information necessary for the loader to complete the linkage between modules when they are loaded into memory at execution time.
- Values that may be specified in byte, halfword, word, and doubleword lengths.
- Instructions that are automatically aligned on word boundaries.
- The COM directive, which allows the user to define instructions and table areas.
- Standard procedures that provide mnemonic repertories for processing instructions available with the various hardware options.
- TEXTC and TEXT directives, which simplify coding of output messages and eliminate the need for character counts.

ERROR DETECTION

During an assembly the source program is checked for syntax errors. If any is found, an appropriate notification is given, and the assembly operation continues. Although an assembled program containing errors generally cannot be executed, the assembler continues to the end of the program in order to locate any additional errors in the same run.

PROGRAM OPERATION

The Symbol assembler can operate as a stand-alone processor or under control of the Xerox Sigma Basic Control Monitor or Batch Processing Monitor. In either case object programs produced by Symbol can be loaded for execution by the Stand-Alone Loader (described in the Stand-Alone Systems OPS Reference Manual, 90 10 53) or by one of the Monitors.

SIGMA MATHEMATICAL LIBRARY

A library of mathematical routines is available to the assembly language programmer. These may be used as stand-alone routines or – under one of the Xerox Monitor systems – as routines available at load time from a peripheral input device. The Mathematical Routines/Technical Manual, 90 09 06, provides a complete description of these routines.

2. LANGUAGE ELEMENTS AND SYNTAX

LANGUAGE ELEMENTS

Input to the assembler consists of a sequence of characters combined to form assembly language elements. These language elements (which include symbols, constants, expressions, and literals) make up the program statements that comprise a source program.

CHARACTERS

The Symbol character set is shown in Table 1.

Table 1. Symbol Character Set

Alphabetic:	A through Z, and $, @, *, :, ← (break character – prints as "underscore")
Numeric:	0 through 9
Special Characters:	Blank
	(Left parenthesis
	) Right parenthesis
	+ Add (or positive value)
	- Subtract (or negative value)
	* Indirect addressing prefix or comments line indicator
	, Comma
	' Constant delimiter (single quotation mark)
	= Introduces a literal
	. Decimal point

The colon is an alphabetic character used in internal symbols of standard Xerox software. It is included in the names of Monitor routines (M:READ) and assembler routines (S:IFR). To avoid conflict between user symbols and those employed by Xerox software, it is suggested that the colon be excluded from user symbols.

SYMBOLS

Symbols are formed from combinations of characters. Symbols provide programmers with a convenient means of identifying program elements so they can be referred to by other elements. Symbols must conform to the following rules:

1. Symbols may consist of from 1 to 8 alphanumeric characters: A-Z, $, @, #, :, ←, 0-9. At least one of the characters in a symbol must be alphabetic. No special characters or blanks can appear in a symbol.

2. The characters $ and $$ may be used in the argument field of a statement to represent the current value of the execution and load location counters, respectively (see Chapter 3); these characters must not be used as label field entries by themselves.

The following are examples of valid symbols:

ARRAY

R1

INTRATE

BASE

7TEMP

#CHAR

SPAYROLL

$ (execution location counter)

The following are examples of invalid symbols:

BASE PAY — Blanks may not appear in symbols.

TWO=2 — Special characters (=) are not permitted in symbols.

CONSTANTS

A constant is a self-defining language element. Its value is inherent in the constant itself, and it is assembled as part of the statement in which it appears.

Self-defining terms are useful in specifying constant values within a program via the EQU directive (as opposed to entering them through an input device) and for use in constructs that require a value rather than the address of the location where that value is stored. For example, the Load Immediate instruction and the BOUND directive both may use self-defining terms:

```
LI,2    57
BOUND   8
```

2, 57, and 8 are self-defining terms

SELF-DEFINING TERMS

Self-defining terms are considered to be absolute (non-relocatable) items since their values do not change when the program is relocated. There are two forms of self-defining terms:

1. The decimal digit string in which the constant is written as a decimal integer constant directly in the instruction:

```
LW,R    HERE+6      "6" is a decimal digit string.
```

   The maximum value of a decimal integer constant is limited to that which can be contained in one word (32 bits).

2. The general constant form in which the type of constant is indicated by a code character, and the value is written as a constant string enclosed by single quotation marks:

```
LW,R    HERE+X'7AF'     "7AF" is a hexadecimal constant representing the decimal value 1967.
```

There are seven types of general constants:

Code	Type
C	Character string constant
X	Hexadecimal constant
O	Octal constant
D	Decimal constant
FX	Fixed-point decimal constant
FS	Floating-point short constant
FL	Floating-point long constant

C: Character String Constant. A character string constant consists of a string of EBCDIC† characters enclosed by single quotation marks and preceded by the letter C:

```
C'ANY CHARACTERS'
```

Each character in a character string constant is allocated eight bits (one byte) of storage.

Because single quotation marks are used as syntactical characters by the assembler, a single quotation mark in a character string must be represented by the appearance of two consecutive quotation marks. For example,

```
C'AB"C"'
```

represents the string

```
AB'C'
```

Character strings are stored four characters per word. The descriptions of TEXT and TEXTC in Chapter 5 provide positioning information pertaining to the character strings used with these directives. In all other usages, character strings must not contain more than sixteen characters. If the string contains less than sixteen characters, the characters are right-justified and a null EBCDIC character(s) fills out the word.

Note: If any constant string enclosed by single quotation marks appears in an object program without one of the type codes listed above, it is assumed to be a character string constant and is processed as if type code C had preceded the string.

X: Hexadecimal Constant. A hexadecimal constant consists of an unsigned hexadecimal number enclosed by single quotation marks and preceded by the letter X:

```
X'9C01F'
```

The assembler generates four bits of storage for each hexadecimal digit. The maximum value of a hexadecimal constant is limited to that which can be contained in one word (32 bits).

The hexadecimal digits and their binary equivalents are as follows:

0 - 0000	8 - 1000
1 - 0001	9 - 1001
2 - 0010	A - 1010
3 - 0011	B - 1011
4 - 0100	C - 1100
5 - 0101	D - 1101
6 - 0110	E - 1110
7 - 0111	F - 1111

Information concerning hexadecimal arithmetic and hexadecimal to decimal conversions is included in the Computer Reference Manuals.

†A table of Extended Binary-Coded Decimal Interchange Codes can be found in the Sigma Computer Reference Manuals.

O: Octal Constant: An octal constant consists of an unsigned octal number enclosed by single quotation marks and preceded by the letter O:

O'7314526'

The maximum value is limited to that which can be contained in one word (32 bits). The size of the constant in binary digits is three times the number of octal digits specified, and the constant is right-justified in its field. For example:

Constant	Binary Value	Hexadecimal Value
O'1234'	001 010 011 100	0010 1001 1100 (29C)

The octal digits and their binary equivalents are as follows:

0 - 000	4 - 100
1 - 001	5 - 101
2 - 010	6 - 110
3 - 011	7 - 111

D: Decimal Constant. A decimal constant consists of an optionally signed value of 1 through 31 decimal digits, enclosed by single quotation marks and preceded by the letter D.

D'735698721' = D'+735698721'

The constant generated by Symbol is of the binary-coded decimal form required for Sigma 7 decimal instructions. In this form, the sign[t] occupies the last digit position, and each digit consists of four bits. For example:

Constant	Value
C'+99'	1001 1001 1100

A decimal constant could be used in an instruction as follows:

```
LW,R     L(D'99')
```

Load (LW) as a literal (L) into register R the decimal constant (D)99.

The value of a decimal constant is limited to that which can be contained in four words (128 bits).

[t] A plus sign is a 4-bit code of the form 1100. A minus sign is a 4-bit code of the form 1101.

FX: Fixed-Point Decimal Constant. A fixed-point decimal constant consists of the following components in the order listed, enclosed by single quotation marks and preceded by the letters FX:

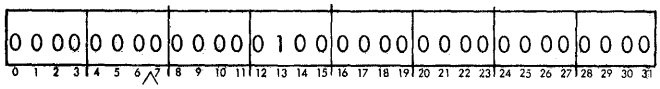
1. An optional algebraic sign.
2. d, d., d.d, or .d, where d is a decimal digit string.
3. An optional exponent:

   The letter E followed optionally by an algebraic sign, followed by one or two decimal digits.
4. A binary scale specification:

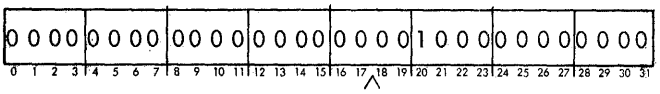
   The letter B followed optionally by an algebraic sign, followed by one or two decimal digits that designate the terminal bit of the integer portion of the constant (i.e., the position of the binary point in the number). Bit position numbering begins at zero.[t]

Parts 3 and 4 may occur in any relative order:

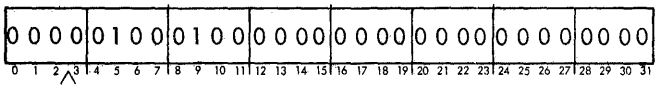
FX'.0078125B6'



FX'1.25E-1B17'



FX'13.28125B2E-2'



The value of a fixed-point decimal constant is limited to that which can be stored in a single word (32 bits). See Example 1.

FS: Floating-Point Short Constant. A floating-point short constant consists of the following components in order, enclosed by single quotation marks, and preceded by the letters FS:

1. An optional algebraic sign.
2. d, d., d.d, or .d, where d is a decimal digit string.
3. An optional exponent.

   The letter E followed optionally by an algebraic sign followed by one or two decimal digits.

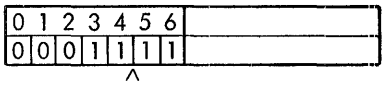
[t] The implied binary point may extend beyond the limits on single words (i.e., FX'1.25B40').

Example 1. Storing Fixed-Point Decimal Constants.

Assume a halfword (16 bits) is to be used for two fields of data; the first field requires seven bits, and the second field requires nine bits.

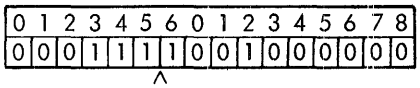
The number FX'3.75B4' is to be stored in the first field. The binary equivalent of this number is 11 ^ 11. The caret represents the position of the binary point. Since the binary point is positioned between bit positions 4 and 5, the number would be stored as

Field 1 Field 2



The number FX'.0625B-2' is to be stored in the second field. The binary equivalent of this number is 0001. The binary point is to be located between bit positions -2 and -1 of field 2; therefore, the number would be stored as

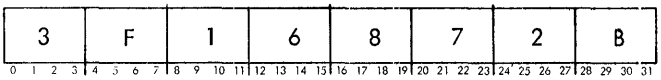
Field 1 Field 2



In generating the second number, Symbol considers bit position -1 of field 2 to contain a zero, but does not actually generate a value for that bit position since it overlaps field 1. This is not an error to the assembler. However, if Symbol were requested to place a 1 in bit position -1 of field 2, an error would be detected since significant bits cannot be generated to be stored outside the field range. Thus, leading zeros may be truncated from the number in a field, but significant digits are not allowed to overlap from one field to another.

Thus, a floating-point short constant† could appear as

FS'5.5E-3'



The value of a floating-point short constant is limited to that which can be stored in a single word (32 bits).

FL: Floating-Point Long Constant. A floating-point long constant consists of the following components in order, enclosed by single quotation marks and preceded by the letters FL:

1. An optional algebraic sign.

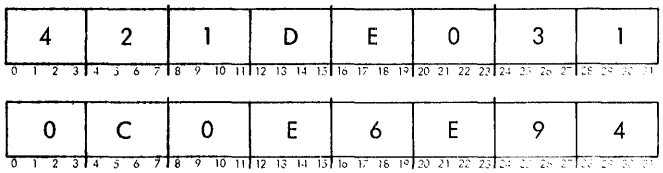
2. d, d., d.d, or .d, where d is a decimal digit string.

3. An optional exponent:

   The letter E followed optionally by an algebraic sign, followed by one or two decimal digits.

Thus, a floating-point long constant† could appear as

FL'2987574839928.E-11'



The value of a floating-point long constant is limited to that which can be stored in two words (64 bits).

LITERALS

A literal is a constant or symbol enclosed by parentheses and preceded by the letter L:

L(-185)	decimal value -185
L(X'5DF')	hexadecimal value 5DF
L(AB)	an address value

or a constant or symbol preceded by an equals sign:

=-185	decimal value -185
=X'5DF'	hexadecimal value 5DF
=AB	an address value

Literals are transformed into references to data values rather than actual values. Literals may be used in any construct that requires an address of a data value rather than the actual value. For example, the Load Word instruction requires the address of the value to be loaded into the register, and use of a literal will satisfy that requirement:

LW,7	L(768)	The value 768 is stored in the literal table and its address is assembled as part of this instruction.

A literal must not be used as a term in a multitermed expression; however, either literal form may be used in an addressing function expression. For example,

BA (HA(L(S + 1)))

is valid.

A literal preceded by an asterisk specifies indirect addressing:

(* = 10).

† Refer to the appropriate Sigma Computer Reference Manual for an explanation of floating-point format.

When a literal appears in a statement, Symbol produces the indicated value, stores the value in the literal table, and assembles the address of that storage location into the statement. The address is assembled as a word address unless the programmer specifies a byte, halfword, or doubleword address (see "Addressing Functions" in Chapter 3). Literals may be used anywhere a storage address value is a valid argument field entry. However, literals may not be used in directives that require previously defined symbols.

During an assembly Symbol generates each literal as a 32-bit value on a word boundary in the literal table. The assembler detects duplicate values and makes only one entry for them in the table. Symbol appends the literal table to the end of the assembled program.

Any of the previously discussed types of constants except floating-point long (FL) may be written as literals:

L(1416)	integer literal
L(C'BYTE')	character string literal
L(X'F0F0')	hexadecimal literal
L(O'7777')	octal literal
L(D'37879')	decimal literal
L(FX'78.2E1B10')	fixed-point decimal literal
L(FS'-8.935410E-02')	floating-point short literal

EXPRESSIONS

An expression is an assembly language element that represents a value. It consists of a single term or a combination of terms (multitermed) separated by arithmetic operators.

A single-termed expression may be any valid symbol reference, a constant, or a literal (symbol references are described later in this chapter).

A multitermed expression must be evaluatable; that is, it must contain only decimal integers, octal or hexadecimal constants, and previously defined symbol references. It must not contain literals, forward references, or external references except for the special case noted later in this chapter under "Forward References" and "External References".

OPERATORS AND EXPRESSION EVALUATION

A single-termed expression, such as 52 or $ or AB, takes on the value of the term involved. A multitermed expression, such as INDX+4, is reduced to a single value by the assembler.

The operators that can appear in a Symbol expression are shown in Table 2.

Table 2. Symbol Operators

Operators	Binding Strength	Function
+	2	Unary plus
-	2	Unary minus
+	1	Integer Add (binary)
-	1	Integer Subtract (binary)

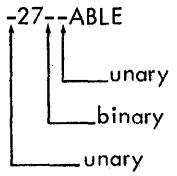
In an expression, operations with the higher binding strength are performed first; those with the same binding strength are performed left to right.

When an address is used as a term in a multitermed expression, the arithmetic operation is restricted to the low-order 19 bits.

The assembler distinguishes between the unary operator (-) and the binary operators in the following manner:

1. An operator preceding an expression may only be a unary operator, as in -27 or +6.

2. The first operator following a term in a multitermed expression must be a binary operator:



If ABLE represents the value 10, the expression would be equivalent to -27-(-10)) = -27 + 10 = -17. If ABLE represents the value -10, the expression would be equivalent to -27-(-10)) = -27-10 = -37.

SYNTAX

Assembly language elements can be combined with computer instructions and assembler directives to form statements that comprise the source program.

STATEMENTS

A statement is the basic component of an assembly language source program; it is also called a source statement, a program statement, or a symbolic line.

Source statements are written on the standard coding form shown in Figure 1.

Xerox Data Systems

XEROX

Xerox Sigma Symbolic Coding Form

PROBLEM NOTHING

PROGRAMMER MVK

IDENTIFICATION 73 80

PAGE 1 OF 1

DATE 7-10-72

LABEL 1 5	COMMAND 10 15	ARGUMENT 20 25 30 35	COMMENTS 37 40 45 50 55 60 65 70 72
*			
*** PROGRAM TO PRINT NOTHING.			
*			
	SYSTEM	SIGS	
	REF	M:LL	
NOTHING	TEXTC	'NOTHING'	
PRINTFPT	GEN,8,24	1,0	
	GEN,1,32	1,0	
	DATA	NOTHING	
START	CAL1,2	PRINTFPT	
	CAL1,9	1	
	END	START	

Figure 1. Xerox Sigma Symbolic Coding Form

FIELDS

The body of the coding form is divided into four fields: label, command, argument, and comments. The coding form is also divided into 80 individual columns. Columns 1 through 72 constitute the active line; columns 73 through 80 are ignored by the assembler except for listing purposes and may be used for identification and a sequence number.

The columns on the coding form correspond to those on a standard 80-column card; one line of coding on the form can be punched into one card.

Symbol provides for free-form symbolic lines; that is, it does not require that each field in a statement begin in a specified column. The rules for writing free-form symbolic lines are:

1. The assembler interprets the fields from left to right: label, command, argument, comments.
2. A blank column terminates any field except the comments field, which is terminated at column 72 on card input or by a carriage return character on paper tape input.
3. One or more blanks at the beginning of a line specify there is no label field entry.
4. The label field entry, when present, must begin in column 1.
5. The command field begins with the first nonblank column following the label field or in the first nonblank column following column 1, if the label field is omitted.

6. The argument field begins with the first nonblank column following the command field. An argument field is designated as blank in either of two ways:

   a. Sixteen or more blank columns follow the command field.

   b. The end of the active line (column 72) is encountered.

7. The comments field begins in the first nonblank column following the argument field or after at least 16 blank columns following the command field, when the argument field is empty.

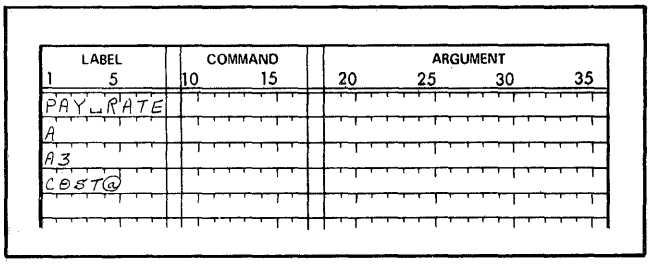
ENTRIES

A source statement may consist of one to four entries written on a coding sheet in the appropriate fields: a label field entry, a command field entry, an argument field entry, and a comments field entry.

A label entry (Example 2) is a symbol that identifies the statement in which it appears. The label enables a programmer to refer to a specific statement from other statements within the program.

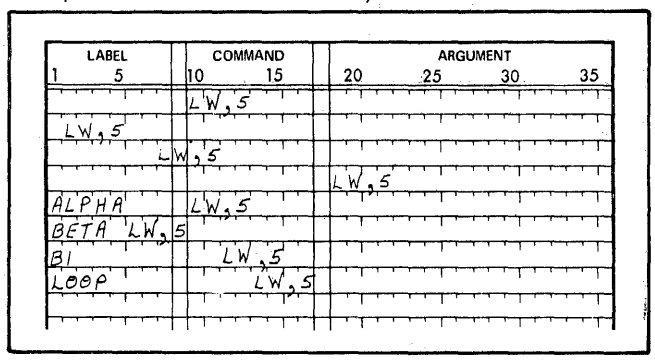
The label of a statement may have the same configuration as an instruction, directive, or intrinsic function without conflict, since Symbol is able to distinguish through context which usage is intended. However, no two statements may have the same label; otherwise, an ambiguous reference would be created. For example, the mnemonic code for the Load Word command is LW. An instruction may be written with LW in the label field, without conflicting with the command LW.

Example 2. Label Field Entry



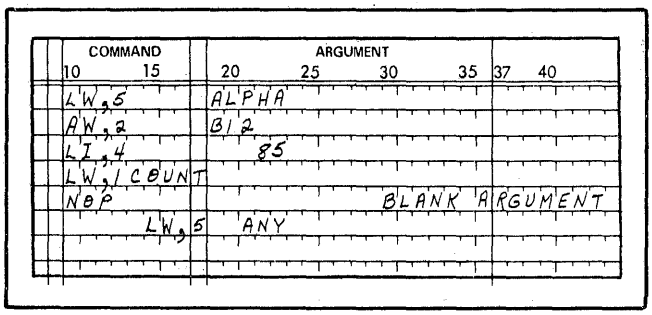
The command entry (Example 3) is a mnemonic code representing a machine instruction or assembler directive specifying the machine operation or assembler function to be performed. A command entry is required in every active line. Thus, if a statement line is entirely blank following the label field or if the command entry is not an acceptable instruction or directive, the assembler declares the statement in error, generates a word of all zeros in the object program, and flags the statement in the assembly listing. The mnemonic codes for machine instructions and the assembler directives recognized by Symbol are listed in Appendixes A and B.

Example 3. Command Field Entry



An argument entry (Example 4) consists of one or more symbols, constants, literals, or expressions separated by commas. The argument entries for machine instructions usually represent such things as storage locations, constants, or intermediate values. Arguments for assembler directives provide the information needed by Symbol to perform the designated operation.

Example 4. Argument Field Entry



A comments entry may consist of any information the user wishes to record. It is read by the assembler and output as part of the source image on the assembly listing. Comments have no effect on the assembly.

COMMENTS LINES

An entire line may be used as a comment by writing an asterisk in column 1. Any EBCDIC character may be used in comments. Extensive comments may be written by using a series of lines, each with an asterisk in column 1.

The assembler reproduces the comment lines on the assembly listing and counts comment lines in making line number assignments (see Chapter 6 for a description of output formats).

PROCESSING OF SYMBOLS

Symbols are used in the label field of a machine instruction to represent its location in the program. In the argument field of an instruction, a symbol identifies the location of an instruction or a data value.

The treatment of symbols appearing in the label or argument field of an assembler directive varies.

DEFINING SYMBOLS

A symbol becomes "defined" by appearing as a label entry. "Defined" means that it is assigned a value. The definition, assigned to the symbol by the assembler, depends on assembly conditions when the symbol is encountered, the contents of the command field, and the current contents of the execution location counter.

Any machine instruction can be labeled; the label is assigned the current value of the execution location counter.

The EQU and COM directives require a label entry; the entry is assigned the value of the symbol or constant in the argument field. A label entry is optional for the following directives: ASECT, CSECT, DATA, DO1, GEN, LOC, ORG, RES, TEXT, and TEXTC. If specified, it is assigned the current value of the execution location counter. For all other directives a label entry is ignored.

The first time a symbol is encountered in the label field of an instruction, or any of the directives mentioned above, it is placed in the symbol table and assigned a value by the assembler. The values assigned to labels naming instructions, storage areas, constants, and control sections represent the addresses of the leftmost bytes of the storage fields containing the named items.

Often the programmer will want to assign values to symbols rather than having the assembler do it. This may be accomplished through use of the EQU directive. A symbol used in the label field of an EQU directive is assigned the value specified in the argument field.

Note: The use of labels is a programmer option, and as many or as few labels as desired may be used. However, since symbol defining requires assembly time and storage space, unnecessary labels should be avoided.

SYMBOL REFERENCES

A symbol used in the argument field of a machine instruction or directive is called a symbol reference. There are three types of symbol references.

PREVIOUSLY DEFINED REFERENCES

A reference made to a symbol that has already been defined is a previously defined reference. All such references are completely processed by the assembler. Previously defined references may be used in any machine instruction or directive.

FORWARD REFERENCES

A reference made to a symbol that has not been defined is a forward reference. A forward reference must not be used as a term in a multitermed expression, with one exception. The exception is that a forward reference may have a constant addend, so that the reference is of the form: reference ± exp or exp + reference. The term exp must be either a positive integer value or an expression that resolves to a positive integer value. Examples of such usage would be

```
        LW,4    HERE-2
          .
          .
HERE    EQU     S
          .
          .
FLAG    EQU     1
          .
          .
        LW,4    FLAG+4+SUM
          .
          .
SUM
```

It should be noted that the negative of a forward reference must not be created by such usage, nor may a forward reference with an addend be used as a literal. For example, if HERE is a forward reference, the usage L(HERE + 2) is illegal.

Any computer instruction may use a forward reference, but only the GOTO, LOCAL, REF, SREF, DEF, GEN, and DATA directives may use forward references. Other directives do not permit the use of forward references.

The argument field entries for most directives must be "evaluatable" expressions; i.e., those that can be evaluated when the assembler encounters them. By definition, such expressions cannot contain forward references.

EXTERNAL REFERENCES

A reference made to a symbol defined in a program other than the one in which it is referenced is an external reference. An external reference must not be used as a term in a multitermed expression, with one exception. The exception is that the external reference may have a constant addend of the same kind and conforming to the same restrictions previously explained under "Forward References".

A program that defines external references must declare them as external by use of the DEF directive. An external definition is output by the assembler as part of the object program, for use by the loader.

A program that uses external references must declare them as such by use of a REF or SREF directive.

A machine instruction containing an external reference is incompletely assembled. The object code generated for such references allows the external references and their associated external definitions to be linked at load time.

After a program has been assembled and stored in memory to be executed, the loader automatically searches the

program library for routines whose labels satisfy any existing external references. These routines are loaded automatically, and interprogram communication is thus completed.

Any computer instruction may contain an external reference; however, external references are not allowed as directive arguments except for REF, SREF, GEN, DATA, and END.

CLASSIFICATION OF SYMBOLS

Symbols may be classified as either local or nonlocal.

A local symbol is one that is defined and may be referenced within a restricted program region. The program region is designated by the LOCAL directive, which also declares the symbols that are to be local to the region.

A symbol not declared as local by use of the LOCAL directive is a nonlocal symbol. It may be defined and referenced in any region of a program, including local symbol regions.

The same symbol may be both nonlocal and local, in which case the nonlocal and local forms identify different program elements.

SYMBOL TABLE

The value of each defined symbol is stored in the assembler's symbol table. Each value has a value type associated with it, such as relocatable address, integer, external reference. Some types require additional information. For example, relocatable addresses, which are entered as 19-bit offsets from the program section base, require the intrinsic resolution of the symbol (see Chapter 3 for a discussion of intrinsic resolution).

When the assembler encounters a symbol in the argument field, it refers to the symbol table to determine if the symbol has already been defined. If it has, the assembler obtains from the table the value and attributes associated with the symbol, and is able to assemble the appropriate value in the statement.

If the symbol is not in the table, it is assumed to be a forward reference. Symbol enters the symbol in the table, but does not assign it a value. When the symbol is defined later in the program, Symbol assigns it a value and designates the appropriate attributes. If any undefined symbols remain in the symbol table after assembly is completed, Symbol considers them to be unknown and produces an appropriate error message.

ABSOLUTE AND RELOCATABLE VALUES

The value of a symbol or expression may be absolute or relocatable. An absolute value, which is assigned at assembly time, is the same value that will be used by the program at execution time. A relocatable value, on the other hand, may be altered by the loader at execution time.

SYMBOL VALUES

A symbol is assigned an absolute value by one of the following methods:

1. By equating the symbol to an absolute numeric quantity:

```
SUM         EQU         2
```

   SUM is assigned the absolute value 2.

2. By equating the symbol to an absolute symbol:

```
A           EQU         -10

ANSWER      EQU         A
```

   ANSWER is assigned the absolute value -10.

3. By equating the symbol to the difference of two relocatable symbols:

```
TAB         DATA        1,2,3

ENDTAB      RES         0

LTAB        EQU         ENDTAB-TAB
```

The value of an absolute symbol does not change, even if it is part of a relocatable program (a program that can be executed anywhere in memory).

A symbol has a relocatable value unless declared absolute as described above. The value of a relocatable symbol may be altered by the loader when the symbol is a part of a relocatable program.

EXPRESSION VALUES

An absolute expression may consist of either a single absolute term or a combination of absolute terms. An absolute term is a hexadecimal, octal, or decimal integer. Note that D, C, FX, FS, and FL constant types are not permitted in expressions.

A relocatable expression may consist of either a single relocatable term or a combination of relocatable terms.

The mode of an expression combining absolute terms with relocatable terms is determined as shown in Example 5.

When the assembler evaluates expression, it determines whether the expression value is relocatable or absolute. Each term in the expression has a relocatable or absolute status value: 1 = relocatable; 0 = absolute. The assembler scans the expression from left to right combining status

values according to the operators in the expression. Symbol allows only two operators in expressions: + and -. At no time during the scan may the accumulation of status values exceed 1.

The resulting status value at the end of the scan indicates whether the expression is relocatable or absolute. An absolute expression is unaffected by the relocation of a program at execution time. A relocatable expression, however, may be altered by the loader at execution time.

Example 5. Expressions Using + and - Operators

Assume R1, R2, and R3 are relocatable terms and A1 and A2 are absolute terms.

Expression:	R1±A1	
Status:	1 0	Legal, relocatable
Accumulation:	1	
Expression:	R1-R2-R3	
Status:	1 1 1	Illegal, diagnostic error
Accumulation:	0 -1	
Expression:	R1-R2+A1	
Status:	1 1 0	Legal, absolute
Accumulation:	0 0	
Expression:	R1-R2+R3-A1+A2	
Status:	1 1 1 0 0	Legal, relocatable
Accumulation:	0 1 1 1	
Expression:	R1+R2	
Status:	1 1	Illegal, diagnostic error
Accumulation:	2	
Expression:	R1+R2-R3	
Status:	1 1 1	Illegal, diagnostic error
Accumulation	2 1	
Expression:	A1±A2	
Status:	0 0	Legal, absolute
Accumulation:	0	

3. ADDRESSING

Sigma computer addressing techniques require a register designation and an argument address which may specify indexing and/or indirect addressing. The programmer may write addresses in symbolic form, and the assembler will convert them to the proper equivalents.

RELATIVE ADDRESSING

Relative addressing is the technique of addressing instructions and storage areas by designating their locations in relation to other locations. This is accomplished by using symbolic rather than numeric designations for addresses. An instruction may be given a symbolic label such as LOOP, and the programmer can refer to that instruction anywhere in his program by using the symbol LOOP in the argument field of another instruction. To reference the instruction following LOOP, he can write LOOP+1; similarly, to reference the instruction preceding LOOP, he can write LOOP-1.

An address may be given as relative to the location of the current instruction even though the instruction being referenced is not labeled. The execution location counter, described later in this chapter, always indicates the location of the current instruction and may be referenced by the symbol $. Thus, the construct $+8 specifies an address eight units greater than the current address, and the construct $-4 specifies an address four units less than the current address.

ADDRESSING FUNCTIONS

Intrinsic functions are functions built into the assembler. Certain of these functions concerned with address resolution are discussed here. Literals, another intrinsic function, were discussed in Chapter 2; other intrinsic functions are explained in Chapter 5.

Intrinsic functions, including those concerned with address resolution, may or may not require arguments. When an argument is required for an intrinsic function, it is always enclosed in parentheses.

A symbol whose value is an address has an intrinsic address resolution assigned at the time the symbol is defined. Usually this intrinsic resolution is the resolution currently applicable to the execution location counter. The addressing functions BA, HA, WA, and DA (explained later) allow the programmer to specify explicitly an intrinsic address resolution other than the one currently in effect.

Certain address resolution functions are applied unconditionally to an address field after it is evaluated. The choice of functions depends on the instruction involved. For instructions that require values rather than addresses (e.g., LI, MI, DATA), no final addressing function is applied. For instructions that require word addresses (e.g., LW, STW, LB, STB, LH, LD), word address resolution is applied. Thus, the assembler evaluates LW,3 ADDREXP as if it were LW,3 WA(ADDREXP). Similarly, instructions that require byte addressing (e.g., MBS) cause a final byte addressing resolution to be applied to the address field.

More information on address resolution is given after the explanation of intrinsic addressing functions, which follows.

$,$$ Location Counters

The symbols $ (current value of execution location counter) and $$ (current value of load location counter) indicate that the current value of the appropriate location counter is to be generated for the field in which the symbol appears (see Example 6).

The current address resolution of the counter is also applied to the generated field. This resolution may be changed by the use of an addressing function.

Example 6. $,$$ Functions

```
            .
            .
            .
A           EQU     $       Equate A to the current
                            value of the execution
                            location counter.
            .
            .
            .
Z           EQU     $$      Equate Z to the current
                            value of the load loca-
                            cation counter.
            .
            .
            .
TEST        BCS,3   $+2     Branch to the location
                            specified by the current
                            execution location counter
                            +2 if the condition code
                            and value 3 compare 1's
            .               anyplace.
            .
            .
```

BA Byte Address

The byte address function (Example 7) has the format

BA(address expression)

where "BA" identifies the function, and "address expression" is the symbol or expression that is to have byte address resolution when assembled. If "address expression" is a constant, the value returned is the constant itself.

Example 7. BA Function

```
        :
Z       LI,3    BAL(L(48))      The value 48 is stored
                                in the literal table
                                and its location is as-
                                sembled into this argu-
                                ment field as a byte
                                address.
        :
AA      LI,5    BA($)           The current execution
                                location counter ad-
                                dress is evaluated as
                                a byte address for this
                                statement.
        :
```

HA Halfword Address

The halfword address function (Example 8) has the format

HA(address expression)

where "HA" identifies the function, and "address expression" is the symbol or expression that is to have halfword address resolution. If "address expression" is a constant, the value returned is the constant itself.

Example 8. HA Function

```
        :
Z       CSECT                   Declares control sec-
                                tion Z. Both location
                                counters are initialized
                                to zero. Z is impli-
                                citly defined as a word
                                resolution address.
Q       EQU     HA(Z+4)         Equates Q to a half-
                                word address of Z+4
                                (words).
        :
```

WA Word Address

The word address function (Example 9) has the format

WA(address expression)

where "WA" identifies the function, and "address expression" is the symbol or expression that is to have word address resolution when assembled. If "address expression" is a constant, the value returned is the constant itself.

Example 9. WA Function

```
        :
A       ASECT                   Declares absolute sec-
                                tion A and sets its loca-
                                tion counters to zero.
        LW,3    Z1              Assembles instruction to
                                be stored in location 0.
B       LW,4    Z2              Assigns the symbol B the
                                value 1, with word ad-
                                dress resolution.
        :
C       EQU     BA(B)           Equates C to the value of
                                B with byte address
                                resolution.
        :
F       EQU     WA(C)           Equates F to the value of
                                C, with word address
                                resolution.
        :
```

DA Doubleword Address

The doubleword address function (Example 10) has the format

DA(address expression)

where "DA" identifies the function, and "address expression" is the symbol or expression that is to have doubleword address resolution when assembled. If "address expression" is a constant, the value returned is the constant itself.

Example 10. DA Function

```
        :
LI,5    DA(L(ALPHA))            The symbol ALPHA is
                                stored in the literal table
                                and its location is assem-
                                bled into this statement
                                as a doubleword address.
        :
```

ADDRESS RESOLUTION

To the assembler an address represents an offset from the beginning of the program section in which it is defined.

Consequently, the assembler maintains in its symbol table not only the offset value, but an indicator that specifies whether the offset value represents bytes, words, halfwords, or doublewords. This indicator is called the "address resolution".

Address resolution is determined at the time a symbolic address is defined, in one of two ways:

1. Explicitly, by specifying an addressing function.

2. Implicitly, by using the address resolution of the execution location counter. (The resolution of the execution location counter is set by the ORG or LOC directives. If neither is specified, the address resolution is word.)

The resolution of a symbolic address affects the arithmetic performed on it. If A is the address of the leftmost byte of the fifth word, defined with word resolution, then the expression A + 1 has the value 6 (5 words + 1 word). If A is defined with byte resolution, then the same expression has the value 21 (20 bytes + 1 byte). See Example 11.

Forward and external references with addends are considered to be of word resolution when used without a resolution function in a generative statement or in an expression. Thus, a forward or external reference of the form

reference + 2

is implicitly

WA(reference + 2)

Symbol restricts the number of nested resolution functions and addends that may be applied to a forward or external reference with an addend. Only one such change of address resolution may be made. For example, the following usage of a forward reference is permissible:

BA(2+WA(reference))

while the following usage cannot be processed by Symbol and will be flagged as an error:

WA(BA(2 + WA(reference)))

Similarly, once a forward or external reference has been given an addend followed by a change of resolution, it may not be given another addend. For example, the following forward reference usage will also be flagged as an error:

(BA(2 + WA(reference)) + 1

LOCATION COUNTERS

A location counter is a memory cell the assembler uses to record the storage location it assigned last and, thus, what location it should assign next. Each program has two location counters associated with it during assembly: the

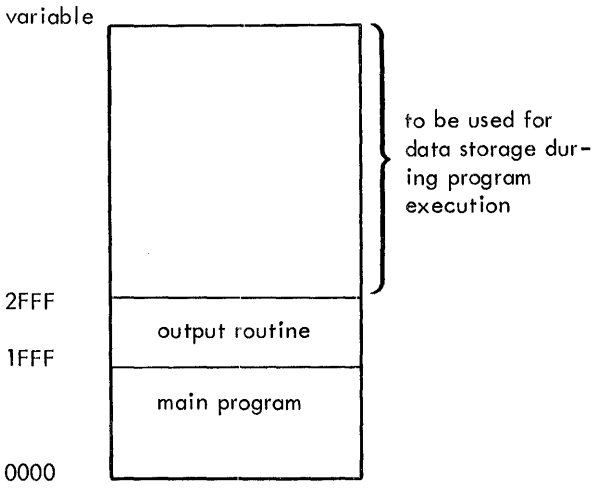
Example 11. Address Resolution

Location		Generated Code				
00000				CSECT ORG	0	Sets value of location counters to zero with word resolution.
00000		FFFB	A	GEN, 16	-5	Defines A as 0 with word resolution.
00000	2	0004	B	GEN, 16	4	Defines B as 0 with word resolution.
00001		0000		GEN, 16	BA(A)	Generates 0 with byte resolution.
00001	2	0002		GEN, 16	BA(B)	Generates 2 with byte resolution.
00002		0001		GEN, 16	HA(B)	Generates 1 with halfword resolution.
00002	2			ORG, 1	$	Sets value of location counters to 10 with byte resolution.
00002	2	FFFF	F	GEN, 16	-1	Defines F as 10 resolution.
00003		000A		GEN, 16	F	Generates 10 with byte resolution.
00003	2	000B		GEN, 16	F+1	Generates 11 with byte resolution.
00004		0002		GEN, 16	WA(F)	Generates 2 with word resolution.
00004	2	0002		GEN, 16	WA(F+1)	Generates 2 with word resolution.
00005		0008		GEN, 16	BA(WA(F+1))	Generates 8 with byte resolution.
00005	2	0003		GEN, 16	WA(F)+1	Generates 3 with word resolution.
00006		000C		GEN, 16	BA(WA(F)+1)	Generates 12 with byte resolution.
00006	2	000D		GEN, 16	BA(WA(F)+1)+1	Generates 13 with byte resolution.

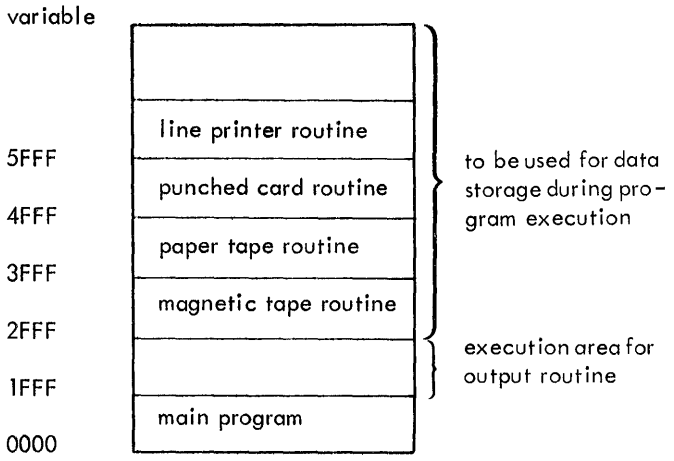
load location counter (referenced symbolically as $$) and the execution location counter (referenced symbolically as $). The load location counter contains a location value relative to the origin of the source program. The execution location counter contains a location value relative to the source program's execution base.

Essentially, the load location counter provides information to the loader that enables it to load a program or subprogram into a desired area of memory. The execution location counter, on the other hand, is used by the assembler to derive the addresses for the instructions being assembled. To express it another way, the execution location counter is used in computing the locations and addresses within the program, and the load location counter is used in computing the storage locations where the program will be loaded prior to execution.

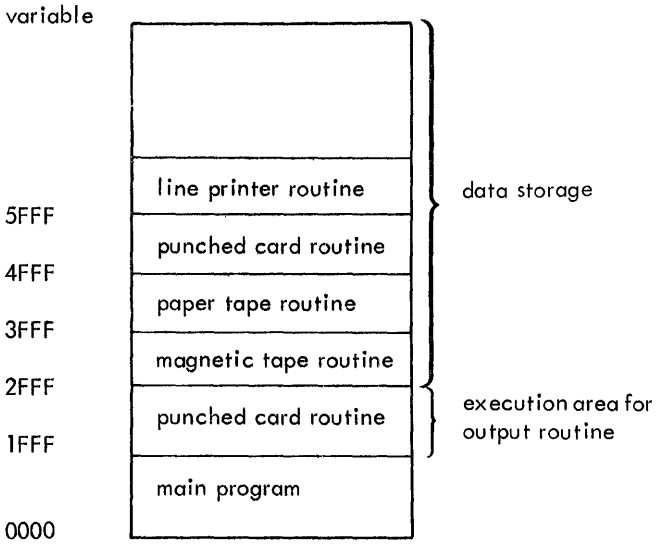
In the "normal" case both counters are stepped together as each instruction is assembled, and both contain the same location value. However, the ORG and LOC directives make it possible to set the two counters to different initial values to handle a variety of programming situations. The load location counter is a facility that enables systems programmers to assemble a program that must be executed in a certain area of core memory, load it into a different area of core, and then, when the program is to be executed, move it to the proper area of memory without altering any addresses. For example, assume that a program provides a choice of four different output routines: one each for paper tape, magnetic tape, punched cards, or line printer. In order to execute properly, the program must be stored in core as follows:



Each of the four output routines would be assembled with the same initial execution location counter value of 1FFF but with different load location counter values. At run time this would enable all the routines to be loaded as shown below.

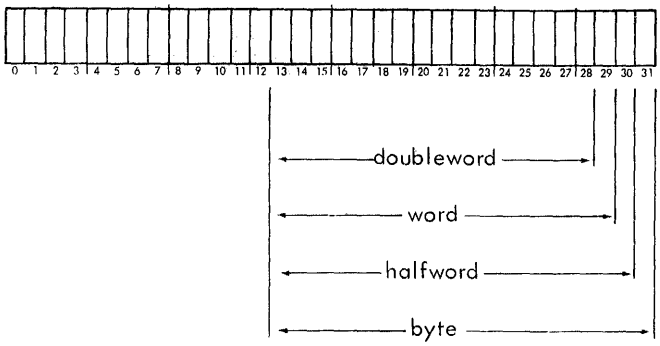


When the main program has determined which output routine is to be used, during program execution, it moves the routine to the execution area. No address modification to the routine is required since all routines were originally assembled to be executed in that area. If the punched card output routine were selected, storage would appear as:

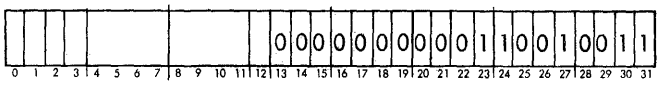


The user should not assume from this example that the execution location counter must be controlled in the manner indicated in order for a program to be relocated. By properly controlling the loader and furnishing it with a "relocation bias", any Symbol program, unless the programmer specifies otherwise, can be relocated into a memory area different from the one for which it was assembled. Most relocatable programs are assembled relative to location zero. To assemble a program relative to some other location, the programmer should use an ORG directive to designate the program origin. This directive sets both location counters to the same value. More information on program sectioning and relocatability is given at the end of this chapter.

Each location counter is a 19-bit value that the assembler uses to construct byte, halfword, word, and doubleword addresses:



Thus, if a location counter contained the value



it could be evaluated as follows:

Resolution	Hexadecimal Value
Byte	193
Halfword	C9
Word	64
Doubleword	32

The address resolution option of the ORG and LOC directives allows the programmer to specify the intrinsic resolution of the location counters. Word resolution is used as the intrinsic resolution if no specification is given. Address functions, as previously explained, are provided to override this resolution.

SETTING THE LOCATION COUNTERS

At the beginning of an assembly, Symbol automatically sets the value of both location counters to zero. The user can reset the location values for these counters during an assembly with the ORG and LOC directives. The ORG directive sets the value of both location counters. The LOC directive sets the value of only the execution location counter.

ORG Set Program Origin

The ORG directive (Examples 12 and 13) sets both location counters to the location specified. This directive has the form

label	command	argument
[label]	ORG[,n]	location

where

label may be any valid symbol. Use of a label is optional. When present, it is defined as the value "location" and is associated with the first byte of storage following the ORG directive.

n may be a constant, symbol, or expression whose value is 1, 2, 3, 4, or 8, specifying the address resolution for both counters as byte, halfword, word, or doubleword, respectively. If n is omitted, word resolution is assumed.

location may be relocatable or an evaluatable expression resulting in a positive integer value.

The address resolution option of ORG may be used to change the intrinsic resolution specification to byte, halfword, or doubleword resolution. Thereafter, whenever intrinsic resolution is applicable, it will be that designated by the most recently encountered ORG directive. For example, whenever $ or $$ are encountered, the values they represent are expressed according to the currently applicable intrinsic resolution.

LOC Set Program Execution

The LOC directive (Example 14) sets the execution location counter ($) to the location specified. It has the form

label	command	argument
[label]	LOC [,n]	location

where

label is any valid symbol. Use of a label is optional. When present, it is defined as the value of "location" and is associated with the first byte of storage following the LOC directive.

n may be a constant, symbol, or expression whose value is 1, 2, 4, or 8, specifying the address resolution for the execution location counter as byte, halfword, word, or doubleword, respectively. If n is omitted, word resolution is assumed.

location may be relocatable or an evaluatable expression resulting in a positive integer value.

Except that it sets only the execution location counter, the LOC directive is the same as ORG.

BOUND Advance Location Counters to Boundary

The BOUND directive (Example 15) advances both location counters, if necessary, so that the execution location counter is a byte multiple of the boundary designated. The form of this directive is

label	command	argument
	BOUND	boundary

where "boundary" may be any evaluatable expression resulting in a positive integer value that is a power of 2. Halfword addresses are multiples of two bytes, fullword addresses are multiples of four bytes, and doubleword addresses are multiples of eight bytes.

Example 12. ORG Directive

```
        :
AA      ORG     8           This directive sets the location counters to 8 and assigns that location to the
                            label AA.

        LW,2    INDEX       This instruction is assembled to be loaded into the location defined as AA.
                            Thus, the effect is the same as if the ORG directive had not been labeled and
                            the label AA had been written with the LW instruction (i.e., AA LW,2 INDEX).
        :
```

Example 13. ORG Directive

```
        :
Z       CSECT               Designates section Z and sets the location counters to zero.

        ORG     Z + 4       Sets the location counters to Z + 4 with word resolution.
        :
A       LW,4    ANY         Assembles ANY with word resolution, and defines A with word resolution.
        :
        MBS,0   B           Forces a byte address. The type of address required by the command overrides
                            the intrinsic resolution of the symbol.

        LI,4    BA(ANY)     Assembles the symbol ANY as a byte address.
        :
```

Example 14. LOC Directive

```
        :
PDQ     ASECT

        ORG     100         Sets the execution location counter and load location counter to 100.

        LOC     1000        Sets the execution location counter to 1000. The load location counter
                            remains at 100.
        :
```

Subsequent instructions will be assembled so that the object program can be loaded anywhere in core relative to the original origin of the program. For example, a relocation bias of 500 will cause the loader to load the program at 600 (500 + 100). However, the program will execute properly only after it has been moved to location 1000.

Example 15. BOUND Directive

```
        BOUND 8             Sets the execution location counter to the next higher multiple of 8 if it
                            is not already at such a value.
```

For instance, the value of the execution location counter for the current section might be 3 words (12 bytes). This directive would advance the counter to 4 (16 bytes), which would allow word and doubleword, as well as byte and halfword, addressing.

When the BOUND directive is processed, the execution location counter is advanced to a byte multiple of the boundary designated and then the load location counter is advanced the same number of bytes. When the BOUND directive results in the location counters being advanced, zeros are generated in the byte positions skipped.

RES Reserve an Area

The RES directive (Example 16) enables the user to reserve an area of core memory.

label	command	argument
[label]	RES[,n]	u

where

label is any valid symbol. Use of a label is optional. When present, the label is defined as the current value of the execution location counter and identifies the first byte of the reserved area.

n is an evaluatable expression designating the size in bytes of the units to be reserved. The value of n must be a positive integer. Use of n is optional; if omitted, its value is assumed to be four bytes.

u is an evaluatable expression designating the number of units to be reserved. The value of u may be a positive or negative integer.

When Symbol encounters a RES directive, it modifies both location counters by the specified number of units.

Example 16. RES Directive

```
      .
      .
      .
      ORG     100       Sets location counters to 100.

A     RES,4   10        Defines symbol A as location
                        100 and advances the location
                        counters by 40 bytes
                        (10 words) changing them
                        to 110.

      LW,4    VALUE     Assigns this instruction the
                        current value of the location
                        counters; i.e., 110.
      .
      .
      .
```

PROGRAM SECTIONS

A Symbol object program may consist of one or two program sections: one relocatable and/or one absolute section. Sectioning is the arbitrary grouping of areas of a program into logical divisions, such as specifying one section for the main program and one for subroutines or data.

It is usually desirable to assemble a symbolic program section without allocating it to a particular memory area of starting location. When a program section can be executed independently of its origin, that is, independently of where it is physically located within the computer, it is called a relocatable program section. Relocatable program sections are frequently assembled relative to location zero; that is, they are assembled as if the first instruction would be stored at location zero. Subsequent instructions are assembled relative to the beginning location of the program.

When a relocatable section is loaded into core to be executed, the user may specify the beginning location of the area where the section is to be stored, and an appropriate value (called a relocation bias) is added to the address portion of each relocatable instruction in the program. For example, a relocatable section assembled relative to location zero may be loaded beginning at location 1000; then, the value 1000 is the relocation bias for that section. To illustrate, assume a section is assembled relative to zero:

```
Location  Instruction
   .         .
   .         .
   .         .
100       B       ALPHA     Branch to location
   .         .               ALPHA.
   .         .
   .         .
120       LW,4    BETA      Load register 4 with
   .         .               contents of BETA.
   .         .
   .         .
```

When these instructions are assembled, the branch instruction in location 100 will specify a transfer to location 120. If this program is loaded with a relocation bias of 1500, the branch instruction would be stored at 1600 and would specify a transfer to location 1620.

Programs are generally relocatable; however, provision is made for an absolute or nonrelocatable section which is useful for such purposes as storing instructions to be executed in the event of an interrupt.

ASECT Absolute Section

CSECT Control Section

ASECT and CSECT are the two directives provided for program sectioning. ASECT declares an absolute control section so that generative statements will be assembled for loading into absolute locations. The location counters are set to absolute zero. CSECT declares a relocatable control section so that generative statements will be assembled for loading into relocatable locations. The location counters are set to relocatable zero.

The program section directives have the form

label	command	argument
[label]	ASECT	
[label]	CSECT	[value]

where "label" is the name by which the section is identified. For both ASECT and CSECT a label is optional, and any valid symbol may be used. The "label" must not be an external reference.

The "value", if specified, must be between 0 and 3. This value indicates the type of memory protection to be associated with the control section. If "value" is omitted, zero is assumed.

Once a program section has been specified, it is effective until another is specified. If a program section is not specified when the assembly begins, Symbol arbitrarily designates an unlabeled, relocatable section and assembles the program accordingly.

At the time a program section (Examples 17 and 18) is originally declared, both location counters are set to zero and their address resolution specifications are word addressing. The address value for either or both of the counters may be altered by an ORG or LOC directive. Thus, ASECT and CSECT directives are often followed immediately by an ORG and/or LOC directive to specify the location of the first byte of the section (see Examples 13 and 14).

Example 17. Program Sectioning

```
INTERPT1        ASECT
```

This statement indicates that subsequent instructions are to be assembled with absolute addresses. The section is identified as INTERPT1. An ORG directive should follow the ASECT statement to designate the absolute address to which the location counter is set if it is to be a value other than zero.

Example 18. Program Sectioning

```
:
TEST CSECT      Declares a relocatable section
                identified as TEST.

                Instructions assembled as part
                of section TEST.
:
LAST            Last instruction in section TEST.

NEW ASECT       Declares a different section,
                identified as NEW and assem-
                bled with absolute addresses.

                Instructions assembled as part
                of section NEW.
:
ORG LAST        Resumes assembling section TEST.
```

4. INSTRUCTIONS

Sigma computer instructions (see Example 19) may be written in symbolic code and combined with other assembly language elements to form symbolic instruction statements.

The four fields of a symbolic instruction statement are:

Field	Contents
label	Any valid symbol. Use of a label entry is optional; when present, the label symbol may also appear in the argument field of other instructions and directives.
command	Any mnemonic operation code listed in Appendix B. The entry may consist of several subfields, the first of which is always the operation mnemonic code. The subsequent subfields may be a register expression, a count expression, or a value expression, depending on the requirements of the particular instruction.
argument	One or more subfields such as an address expression, an indirect addressing designator, or a displacement expression, depending on the requirements of the specific instruction.
comments	Any remark explaining the specific purpose of the statement of the overall function of the program.

Machine language instructions are automatically aligned on word boundaries by the assembler. The address expressions in the argument fields of these instructions are assembled according to the dictates of the specific instruction and the dictates of any addressing functions in the argument. (See Example 13 in Chapter 3.)

Appendix B contains a summary of machine language instruction mnemonics specifying the requirements of each field. The Xerox Sigma Computer Reference Manuals contain complete descriptions of these instructions.

Example 19. Sigma 5-7 Instructions

label	command	argument	comments
L1	LW,4	HOLD	Load Word from location HOLD into register 4.
L2	LW,4	HOLD, 2	Indexed Load Word instruction using register 2 as an index register.
L3	LW,4	*HOLD, 2	A Load Word instruction that specifies both indexing and indirect addressing.
L4	LI,3	X'F3E'	Load the hexadecimal value F3E from the argument field into register 3.
L5	AW,12	L(32)	Add the decimal value 32 to the contents of register 12.
L6	B	LOOP	Branch unconditionally to location LOOP.

Although the general registers and index registers are specified only by digits in these examples, they may be arithmetic expressions whose values are 0-15 for general registers and 0-7 for index registers. They also may be symbols that have been assigned values within that range (i.e., X1 EQU1).

5. SYMBOL DIRECTIVES

A directive is a command to the assembler that can be combined with other language elements to form statements. Directive statements, like instruction statements, have four fields: label, command, argument, and comments.

An entry in the label field is required for two directives: EQU and COM. EQU equates the symbol in the label field to the value of the expression in the argument field. The label field entry for COM identifies the command that COM generates.

Optional labels for the directives ORG and LOC are defined as the value to which the execution location counter is set by the directive.

If any of the directives DATA, GEN, RES, TEXT, or TEXTC are labeled, the label is assigned the current value of the execution location counter and identifies the first word of the area generated or specified by the directive. These directives also alter both location counters, according to the contents of the argument field.

Labels for the directives ASECT, CSECT, and DO1 identify the first word of the area affected by the directives. These directives are nongenerative and do not alter the location counters.

For the directives BOUND, DEF, END, GOTO, LOCAL, PAGE, REF, SREF, and SYSTEM, a label field entry is ignored. The symbol in the label field is not defined, and, therefore, may not be referenced unless it is the target label in a GOTO search.

The command field entry is the directive itself. For some directives this field may consist of two subfields (e.g., GOTO ,k), in which case the directive must be in the first subfield, followed by the other entry.

Argument field entries vary and are defined in the discussion of each directive.

A comments field entry is optional.

The Symbol assembly language includes these directives:

Assembly Control

SYSTEM	ORG†	ASECT†
END	LOC†	CSECT†
DO1	BOUND†	
GOTO	RES†	

Symbol Manipulation

LOCAL	REF
EQU	SREF
DEF	

Data Generation		Listing Control
GEN	TEXT	PAGE
COM	TEXTC	
DATA		

In the formats that follow, brackets indicate optional items.

ASSEMBLY CONTROL

SYSTEM Call System

SYSTEM directs the assembler to define the subset of computer instructions that are to be valid during this portion of the assembly. This directive has the form

label	command	argument
	SYSTEM	name

where "name" identifies the instruction set, and must be one of the following:

Name	Instruction Set
SIG7	Basic Sigma 7.
SIG7F	Sigma 7 with Floating-Point Option.
SIG7D	Sigma 7 with Decimal Option.
SIG7P	Sigma 7 with Privileged Instructions.
SIG7FD	Sigma 7 with Floating-Point and Decimal Option.
SIG7FP	Sigma 7 with Floating-Point Option and Privileged Instructions.
SIG7DP	Sigma 7 with Decimal Option and Privileged Instructions.
SIG7FDP	Sigma 7 with Floating-Point, Decimal Option, and Privileged Instructions.

†Discussed in Chapter 3, "Addressing".

Name	Instruction Set
SIG6	Basic Sigma 6 (decimal instructions are included).
SIG6F	Sigma 6 with Floating-Point Option.
SIG6P	Sigma 6 with Privileged Instructions.
SIG6FP	Sigma 6 with Floating-Point Option and Privileged Instructions.
SIG5	Basic Sigma 5.
SIG5F	Sigma 5 with Floating-Point Option.
SIG5P	Sigma 5 with Privileged Instructions.
SIG5FP	Sigma 5 with Floating-Point Option and Privileged Instructions.

None of the instruction sets omits any of the intrinsic commands or functions. Symbol assumes a default specification of SIG7FDP when SYSTEM is not specified.

END End Assembly

The END directive (Example 20) terminates the assembly of the source program. It has the form

label	command	argument
	END	[exp]

A label field entry is ignored unless it is the target label of a GOTO search. The optional expression in the argument field designates a location to be transferred to after the program has been loaded. Normally, that location contains the first machine language instruction in the program. The expression may be an externally defined symbol (explained later in DEF and REF), in which case the location represented by the symbol exists in a separately assembled program.

As explained later under GOTO, the END directive is processed even when it appears within the range of a GOTO search.

Example 20. END Directive

```
            SYSTEM      SIG7
              .
              .
              .
CONTROL     CSECT
              .
              .
              .
START       LW,5        TEST
              .
              .
              .
            END         START
```

DO1 Iteration Control

The DO1 directive (Example 21) defines the beginning of a single statement assembly iteration loop. It has the form

label	command	argument
[label]	DO1	exp

where

label is any valid label. Use of a label is optional. When present, it is defined as the current value of the execution location counter and identifies the first byte generated as a result of the DO1 iteration.

exp is an evaluatable expression resulting in a positive integer that represents the number of times the line immediately following is to be assembled. There is no limit to the number of times the line may be assembled.

If the expression in the DO1 directive is not evaluatable, Symbol produces an error notification, and processes the DO1 directive as if the expression had been evaluated as zero.

Example 21. DO1 Directive

The statements

```
  .
  .
  .
DO1         3
AW,4        C
  .
  .
  .
```

at assembly time would generate in-line machine code equivalent to the following lines:

```
  .
  .
  .
AW,4        C
AW,4        C
AW,4        C
  .
  .
  .
```

GOTO Conditional Branch

The GOTO directive (Example 22) enables the user to conditionally alter the sequence in which statements are assembled. This directive has the form

label	command	argument
	GOTO[,k]	$label_1$[,...,$label_n$]

where

k is an absolute, evaluatable expression whose value refers to the kth label in the argument field. If k is omitted, 1 is assumed.

$label_i$ are forward references.

A GOTO statement is processed at the time it is encountered during the assembly. Symbol evaluates the expression k and resumes assembly at the line that contains a label corresponding to the kth label in the GOTO argument field. The labels must refer to lines that follow the GOTO directive. If the value of k does not lie between 1 and n, Symbol resumes assembly at the line immediately following the GOTO directive. An error message is generated if k is greater than n.

Although a label on BOUND, DEF, END, GOTO, LOCAL, PAGE, REF, SREF, and SYSTEM is normally ignored by the assembler, it will be recongnized if it is the target label of a GOTO search.

While Symbol is searching for the statement whose label corresponds to the kth label, it operates in a skipping mode during which it ignores all machine language instructions and directives except END and LOCAL. Skipped statements are produced on the assembly listing in symbolic form, preceded by *S*.

If Symbol encounters the END directive before it finds the target label of a GOTO search or if it encounters a LOCAL directive while searching for a local label, it produces an error notification and terminates the assembly.

Example 22. GOTO Directive

```
            :
            :
A           EQU             2
            :
            :
            GOTO, A+2       B, C, D, E, F, G
            :
            :
F
            :
            :
B
            :
            :
E
            :
            :
G
            :
            :
```

Since the expression A + 2 has the value 4, Symbol locates the fourth label in the argument field and resumes assembly at the statement labeled E.

SYMBOL MANIPULATION

LOCAL Declare Local Symbols

As mentioned in Chapter 2, most symbols in a program are "nonlocal" symbols becuase they occur within an implicit nonlocal region. The implicit nonlocal symbol region in a program can be terminated and a new region begun by the LOCAL directive, which has the form

label	command	argument
	LOCAL	[$name_1$, $name_2$, ..., $name_n$]

where the $name_i$ are symbols that are to be local to the current region. Local symbols are syntactically the same as nonlocal symbols. The argument field may be blank, in which case the LOCAL directive (see Example 23) terminates the current local symbol region without declaring any new local symbols.

A label field entry is ignored by the assembler unless it is the target label of a GOTO search.

Any symbols that do not appear in the argument field of this directive retain their original meaning. That is, within a local symbol region only the symbols declared in the LOCAL directive are unique to that region (see Examples 24, 25, and 26).

The local symbol region begins with the first statement (other than comments or another LOCAL) following the LOCAL directive and is terminated by a subsequent use of the LOCAL directive.

Example 23. LOCAL Directive

```
                :
                :
                LOCAL     A, B, C
                LOCAL     R, S, T, U
                LOCAL     X, Y, Z
*COMMENT
START           EQU       S
                :
                :
                LOCAL
```

The three LOCAL directives inform the assembler that the symbols A, B, C, R, S, T, U, X, Y, and Z are to be local to the region beginning with the line START. The final LOCAL directive terminates the local symbol region without declaring any new local symbols.

Example 24. LOCAL Directive

```
            SYSTEM    SIG7
            :
            :
A           CSECT
            :
            :
START       LW, 5     TEST
            :
            :
            LOCAL     TEST      Declares a local symbol
                                region where TEST is local
                                and all others are nonlocal.
            :
            :
            LW, 5     TEST      This TEST does not have
                                the same value as in the
                                statement labeled START.
            :
            :
```

Example 25. LOCAL Directive

```
            .
            .
            .
A           EQU     X'E1'
            .
            .
            .
            LOCAL   A       New A, not the same
                            as A above.
            .
            .
            .
A           EQU     89      Legal, since this is
                            the local A.
            .
            .
            .
B           EQU     A       Defines B as the
                            decimal value 89.
            .
            .
            .
            LOCAL   Z       Terminates current
                            local symbol region
                            and initiates a new
                            region.
            .
            .
            .
Z           EQU     A       Z is equated to the
                            hexadecimal value E1.
            .
            .
            .
```

Example 26. LOCAL Directive

```
            .
            .
            .
ALPHA       ASECT
            ORG     100
            .
            .
            .
S           EQU     T       T and Z must be pre-
X           EQU     Z       viously defined.
            .
            .
            .
            LOCAL   X,Y,Z   Begin a local symbol
                            region where X, Y,
                            and Z are local and
                            all others are nonlocal.
            .
            .
            .
Y           EQU     Z       This Z does not have
                            the same value as the
                            one in the EQU state-
                            ment above.
            .
            .
            .
            LW,2    T       Same T as above, i.e.,
                            a nonlocal symbol.
            .
            .
            .
            LOCAL   A,B,X   End current local sym-
                            bol region and begin
                            a new one where only
                            A, B, and X are local.
            .
            .
            .
            LW,4    Z       This has the same
                            value as the Z that ap-
                            peared in statement
                            X prior to the first
                            LOCAL directive.
            .
            .
            .
X           EQU     N       New definition of X,
                            different from either
                            of the Xs that ap-
                            peared before.
            .
            .
            .
```

EQU Equate Symbols

The EQU directive (Example 27) enables the user to define a symbol by assigning to it the attributes of the expression in the argument field. This directive has the form

label	command	argument
label	EQU	exp

where

label is a valid symbol.

exp is an evaluatable expression whose value is to be associated with "label". The mode (absolute or relocatable) of "exp" is assigned to label.

When EQU is processed by Symbol, "label" is defined as the value of "exp". For example, the statement

```
VALUE   EQU   8+5
```

assigns the absolute value 13 to VALUE, and

```
ALPHA   EQU   S - 10
```

assigns the relocatable value S - 10 to ALPHA.

A symbol defined with an EQU cannot be redefined:

```
A       EQU   X'F'    Legal
        .
        .
        .
A       EQU   O'2'    Illegal because A has already
                      been equated to a value.
```

A symbol appearing in a REF directive (explained below) cannot be used in the argument field of an EQU directive

because the value of such a symbol is not available to the assembler: it is contained (defined) in some other, separately assembled program

Example 27. EQU Directive

```
A       EQU       10          A = 10
        :
        :
B       EQU       A + 4       B = 14
        :
        :
        LW, A     DELTA       Loads the contents of
                              location DELTA into
                              register 10.
```

DEF Declare External Definitions

The DEF directive (Example 28) declares which symbols defined in this assembly may be referenced by other, separately assembled programs. The form of this directive is

label	command	argument
	DEF	$symbol_1$ [, $symbol_2$, . . . , $symbol_n$]

where "symbol" may be any symbolic labels defined within the current program.

A label field entry is ignored unless it is the target label of a GOTO search.

DEF-declared symbols can be used for symbolic program linkage between two or more programs. Such symbols provide access to a program from another program; "access" may be a transfer of control via a branch instruction, or some reference to data storage.

Symbol requires that DEF directives precede any statements that cause code to be generated; this includes all machine-language instructions and the directives BOUND, DATA, DO1, END, GEN, TEXT, and TEXTC. Furthermore, all DEF directives must precede any REF and/or SREF directives.

Example 28. DEF Directive

```
DEF         TAN,SUM,SORT
This statement identifies the labels TAN, SUM, and
SORT as symbols that may be referenced by other programs.
```

REF Declare External Reference

The REF directive (Examples 29 and 30) declares which symbols referred to in this assembly are defined in some other, separately assembled program. The directive has the form

label	command	argument
	REF	$symbol_1$ [, $symbol_2$, . . . , $symbol_n$]

where "$symbol_i$" may be any labels that are to be satisfied at load time by other programs.

A label field entry is ignored unless it is the target label of a GOTO search.

Symbols declared with REF directives can be used for symbolic program linkage between two or more programs. At load time these labels must be satisfied by corresponding external definitions (DEFs) in another program.

REF directives must precede any statements that cause code to be generated; this includes all machine language instructions and directives BOUND, DATA, DO1, END, GEN, TEXT, and TEXTC. REF directives must not precede DEF directives.

Example 29. REF Directive

```
REF         IOCONT, TAPE, TYPE, PUNCH

This statement identifies the labels IOCONT, TAPE,
TYPE, and PUNCH as symbols for which external defi-
nitions will be required at load time.
```

Example 30. REF Directive

```
        REF           Q          Q is an external reference.
        :
        :
B       GEN,16,16     Q,S        The value of an external
                                 reference may be placed
                                 in any portion of a ma-
                                 chine's word.
        :
        :
        LW, 2         Q          Q is an external reference.
        :
        :
        :
```

SREF Secondary External References

The SREF directive is similar to REF and has the form

label	command	argument
	SREF	$symbol_1$[, $symbol_2$, . . . , $symbol_n$]

where the "$symbol_i$" have the same meaning as for REF.

A label field entry is ignored unless it is the target label of a GOTO search.

SREF differs from REF in that REF causes the loader to load routines whose labels it references whereas SREF does not.

Instead, SREF informs the loader that if the routines whose labels it references are in core, then the loader should satisfy the references and provide the interprogram linkage. If the routines are not in core, SREF does not cause the loader to load them; however, it does cause the loader to accept any references within the program to the symbols without considering them to be unsatisfied external references.

Like REF, SREF directives must precede any statements that cause code to be generated and must follow all DEF directives.

DATA GENERATION

GEN Generate a Value

The GEN directive (Examples 31 through 34) produces a hexadecimal value representing the specified bit configuration. It has the form

label	command	argument
[label]	GEN, field list	value list

where

label is any valid symbol. Use of a label is optional. When present, it is defined as the current value of the execution location counter and identifies the first byte generated. The location counters are incremented by the number of words generated.

field list is a list of evaluatable expressions that define the number of bits comprising each field. The sum of the field sizes must be a positive integer value that is a multiple of eight and is less than or equal to 128.

value list is a list of expressions that define the contents of each generated field. This list may contain forward references. The value, represented by the value list, is assembled into the field specified by the field list and is stored in the defined location (see Example 33).

There is a one-to-one correspondence between the entries in the field list and the entries in the value list; the code is generated so that the first field contains the first value, the second field the second value, etc. The value produced by a GEN directive appears on the object program listing as eight hexadecimal digits per line.

External references, forward references, and relocatable addresses may be generated in any portion of a machine word; i.e., an address may be generated in a field that overlaps word boundaries.

A forward reference that does not have a resolution function applied to it is generated with word resolution when it appears in a GEN directive, a DATA directive, or a COM reference line.

Example 31. GEN Directive

```
GEN, 16, 16    -251, 89    Produces two 16-bit hexa-
                           decimal values: FF05 and
                           0059.
```

Example 32. GEN Directive

```
        :
        :
B       EQU         X'FFFFFFFF'
        GEN, 64     B           Produces: 00000000
                                          FFFFFFFF
        .
        :
```

Example 33. GEN Directive

```
        BOUND       4           Specifies word boundary.

LAB     GEN, 8, 8, 8  8, 9, 10  Produces three consecutive bytes; the first is identified as LAB and
                                contains the hexadecimal value 08; the second contains the hexadeci-
                                mal value 09; and the third byte contains the hexadecimal value 0A.
        :
        :
        LW, 5       L(2)        Load register 5 with the literal value 2.
        :
        :
        LB, 3       LAB, 5      Load byte into register 3. LAB specifies the word boundary at which
                                the byte string begins, and the value of the index register (i.e., the
                                value 2 in register 5) specifies the third byte in the string (byte string
                                numbering begins at 0). Thus, this instruction loads the third byte of
                                LAB (the value 0A) into register 3.
```

Example 34. GEN Directive

```
          .
          .
ALPHA     EQU     X'F'          Defines ALPHA as
                                the decimal value
                                15.
BETA      EQU     X'C'          Defines BETA as
          .                     the decimal value
          .                     12.
A         GEN,32 ALPHA+BETA     Defines A as the
                                current location
                                and stores the
                                decimal value 27
                                in 32 bits.
```

In this case, the GEN line is equivalent to

```
A         GEN,32  27
          .
          .
```

COM Command Definition

The COM directive (Examples 35 through 38) enables the programmer to describe subdivisions of computer words and invoke them simply. This directive has the form

label	command	argument
name	COM,field list	value list

where

name is any valid symbol and identifies the command being defined. The "name" must not be a local symbol nor the same as a Sigma machine instruction or Symbol directive.

field list is a list of evaluatable expressions that define the number of bits comprising each field. The sum of the elements in this list must be a positive integer value that is a multiple of eight bits and is less than or equal to 128.

value list is a list of constants or intrinsic functions (see below) that specify the contents of each field.

When the COM directive is encountered, the label, field list, and value list specifications are saved. When the label of the COM directive subsequently appears in the command field of a statement called a "COM reference line", that statement will be generated with the configuration specified by the COM directive.

In Symbol, an asterisk preceding a field list element on the COM definition line specifies that the absence of a corresponding parameter on the COM reference line is to be flagged as an error. See Example 38.

The use of commands defined by a COM is referenced as follows: the COM command definition must precede all references to it.

The COM directive differs from GEN in that Symbol generates a value at the time it encounters a GEN directive, whereas it stores the COM directive and generates a value only when a COM reference line is encountered. If the reference line is labeled, the generated value will be identified by that value.

In Symbol, if a COM directive is to produce four bytes, it will be preceded at reference time by an implicit BOUND,4.

Certain intrinsic functions enable the user to specify in the COM directive which fields in the reference lines will contain values that are to be generated in the desired configuration. These functions are

CF

AF

AFA

CF Command Field

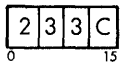
This function (Example 35) refers to the command field list in a reference line of a COM directive. Its format is

CF(element number)

The "CF" specifies the command field, and "element number" specifies which element in the field is being referenced. "Element number" enclosed in parentheses is required. Since a machine language instruction mnemonic or assembler directive must be the first element in the command field on the COM reference line, the element number for the CF function must be two or greater.

Example 35. COM Directive and CF Function

```
          .
          .
BYT       COM,8,8       CF(2), CF(3)
          .
          .
XX        BYT,35,X'3C'
          .
          .
```



The COM directive defines a 16-bit area consisting of two 8-bit fields. It further specifies that data for the first 8-bit field will be obtained from command field 2(CF(2)) of the COM reference line, and that data for the second 8-bit field will be obtained from command field 3(CF(3)). Therefore, when the XX reference line is encountered, Symbol generates a 16-bit value, so that the first eight bits contain the binary equivalent of the decimal number 35 and the second eight bits contain the binary equivalent of the hexadecimal number 3C.

AF Argument Field

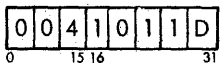
This function (Example 36) refers to the command field list in a reference line of a COM directive. Its format is

AF(element number)

The "AF" specifies the argument field, and "element number" specifies which element in the list of elements in that field is being referenced. "Element number" enclosed in parentheses is required.

Example 36. COM Directive and AF Function

```
          :
          :
XYZ       COM, 16, 16     AF(1), AF(2)
          :
          :
ALPHA     EQU             X'21'
ZZ        XYZ             65,ALPHA+X'FC'
          :
```



Symbol stores the COM definition for later use. When it encounters the ZZ reference line, it references the COM definition in order to generate the correct configuration. At that time, the expression ALPHA+X'FC' is evaluated. AF(1) in the XYZ line refers to 65 in the ZZ line; AF(2) refers to ALPHA+X'FC'.

AFA Argument Field Asterisk

The AFA function (Example 37) determines whether the specified argument in the COM reference line is preceded by an asterisk. The format for this function is

AFA(element number)

where "AFA" identifies the function, and "element number" specifies which element in the argument field of the COM reference line is to be tested. "Element number" is required, and must be enclosed in parentheses. The function produces a value of 1 (true) if an asterisk prefix exists on the argument designated; otherwise, it produces a zero value (false).

DATA Produce Data Value

DATA (Example 39) enables the programmer to represent data conveniently within the symbolic program. It has the form

label	command	argument
[label]	DATA[,f]	$value_1$[,$value_2$,...,$value_n$]

where

label is any valid symbol. Use of a label is optional. When present, it is defined as the current value of the execution location counter and is associated with the first byte generated by the DATA directive. The location counters are incremented by the number of words generated.

Example 37. COM Directive and AFA Function

```
          :
          :
STORE     COM, 1, 7, 4, 4     AFA(1), X'35', CF(2), AF(1)
          :
          :
          STORE, 4            *TOTAL
          :
          :
```

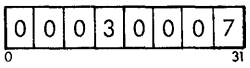
The COM directive defines STORE as a 16-bit area with four fields. The AFA(1) intrinsic function tests whether an asterisk precedes the first element in the argument field of the reference line. The first bit position of the area generated will contain the result of this test. The next seven bits of the area will contain the hexadecimal value 35. The second element in the command field of the reference line will constitute the third field generated, while the first element in the argument field of the reference line will constitute the last field.

When the reference line is encountered, Symbol defines a 16-bit area as follows:

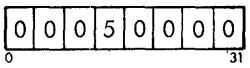
Bit Positions	Contents
0	The value 1 (because the asterisk is present in argument field 1).
1-7	The hexadecimal value 35.
8-11	The value 4.
12-15	The 4-bit value associated with the symbol TOTAL.

Example 38. COM Directive's Error Notification

```
          :
          :
MAP       COM,*16,*16      CF(2),AF(1)
          :
          :
R         MAP,3            7
```




```
          :
          :
X         MAP,5
          :
          :
```



When the first reference line is encountered, Symbol defines a location R and generates a 32-bit data word with the values 3 and 7 in the left and right halfwords, respectively.

When the second reference line is encountered, an error notification is produced because the argument field entry is missing. However, the assembly is not terminated; Symbol will define a location X and generate a 32-bit data word with the values 5 and 0 (for the missing entry) in the left and right halfwords, respectively.

Example 39. DATA Directive

```
          :
          :
MASK1     DATA,1           X'FF'
```

Produces an 8-bit value identified as MASK1.


```
          :
          :
MASK2     DATA,2           X'1EF'
```

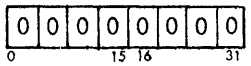
Generates the hexadecimal value 01EF as a 16-bit quantity, identified as MASK2.

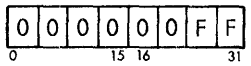

```
          :
          :
BYTE      DATA,3           BA(L(59))
          :
          :
```

The byte address of the literal value 59 is assembled in a 24-bit field, identified as BYTE.

```
TEST      DATA             0,X'FF'
```

Generates two 4-byte quantities; the first contains zeros, and the second, the hexadecimal value 000000FF. The first value is identified as TEST.






```
          :
          :
DT4       DATA,1           X'94',X'CF',X'AB'
```

Generates three 8-bit values, the first of which is identified as DT4.


```
          :
          :
```

f is the field size specification in bytes; f may be any evaluatable expression that results in an integer value in the range $1 \leq f \leq 16$.

$value_i$ are the list of values to be generated. A value may be a multitermed expression or any symbol. An addressing function may be used to specify the resolution of a value when an address resolution other than the intrinsic resolution of the execution location counter is desired.

DATA generates each value in the list into a field whose size is specified by f in bytes. If f is omitted, four bytes are assumed.

Constant values must not exceed those specified under "Constants" in Chapter 2.

TEXT EBCDIC Character String

The TEXT directive (Example 40) enables the user to incorporate messages in his program to be output on some device other than the typewriter via the Monitor's standard output subroutines, or output on the typewriter by some routine other than the Monitor's standard one. This directive has the form

label	command	argument
[label]	TEXT	'cs'

where

label is any valid symbol. Use of a label is optional. When present, it is defined as the current value of the execution location counter and identifies the first byte of the character string generated by the TEXT directive.

'cs' is a character string constant (see Chapter 2).

The character string is assembled in a binary-coded form in a field that begins at a word boundary and ends at a word boundary. The first byte contains the first character of the character string, the second byte contains the second character, etc. If the character string does not require an even multiple of four bytes for its representation, trailing blanks are produced to occupy the space to the next word boundary.

Example 40. TEXT Directive

```
COL1    TEXT    C'VALUE OF X'
```

generates

V	A	L	U
E		O	F
	X		

TEXTC Text with Count

The TEXTC directive (Example 41) enables the user to incorporate messages in a program to be output on the typewriter via the Monitor's standard typewriter output subroutine. This directive has the form

label	command	argument
[label]	TEXTC	'cs'

where "label" and "cs" have the same meanings as for TEXT.

The TEXTC directive provides a byte count of the storage space required for the message. The count is placed in the first byte of the storage area and the character string follows, beginning in the second byte. The count represents only the number of characters in the character string; it does not include the byte it occupies nor any trailing blanks. The maximum number of characters for a single TEXTC directive is 63.

In all other aspects, the TEXTC directive functions in the same manner as the TEXT directive.

Example 41. TEXTC Directive

```
ALPHA   TEXTC   C'VALUE OF X SQUARED'
```

18	V	A	L
U	E		O
F		X	
S	Q	U	A
R	E	D	

LISTING CONTROL

PAGE Begin a New Page

The PAGE directive causes the assembly listing to be advanced to a new page. This directive has the form

label	command	argument
	PAGE	

A label field entry is ignored by the assembler unless it is the target label of a GOTO search. An argument field entry is always ignored.

The PAGE directive is effective only at assembly time. No code is generated for the object program as a result of its use.

6. ASSEMBLY LISTINGS

The Symbol assembler can operate as a stand-alone processor or under control of one of the XDS Monitors – BCM, BPM, or BTM. In all cases the format of the assembly listing is the same.

SYMBOL ASSEMBLY LISTING

The XDS Symbol assembler produces listing lines according to the format shown in Figure 2.

EQUATE SYMBOLS LINE

Each source image line that contains the equate symbols (EQU) directive contains the following information:

NNNNN	Source image line number in decimal.
LLLLL	Value of argument field as a hexadecimal word address.
B	Blank 1, 2, or 3, specifying the byte displacement from word boundary.
or	
XXXXXXXX	Value of argument field as a 32-bit value.
SSS...	Source image.

ASSEMBLY LISTING LINE

Each source image line containing a generative statement, a statement that causes the assembler to generate object code, contains the following information:

NNNNN	Source image line number in decimal.
LLLLL	Current execution location counter to word level in hexadecimal.
B	Blank 1, 2, or 3, specifying the byte displacement from word boundary.
XX XXXX, XXXXXX, XXXXXXXX	Object code in hexadecimal listed in groups of 1 to 4 bytes.
A	Address classification flag:
blank	denotes a relocatable address field.
A	denotes an absolute address field.
F	denotes an address field containing a forward reference.

Print Position	1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38
Equate symbols line	NNNNN or NNNNN L L L L L B S S S... X X X X X X X X S S S...
Assembly listing line	NNNNN L L L L L B X X X X X X X X A S S S...
Ignored source image line	NNNNN * S * S S S...
Error line	* * * E R R O R
Literal listing line	L L L L L X X X X X X X X X
Symbol abort line	S A
Error count line	E C N N N

Figure 2. Symbol Listing Format

X denotes an address field containing an external reference.

N indicates that the object code produced for the source line contains a relocatable item (e.g., an address, a forward reference, or external reference) in some field other than the address field.

SSS... Source image

IGNORED SOURCE IMAGE LINE

The configuration

S

is printed in columns 33-35 for each statement skipped by the assembler during a search for a GOTO label. NNNNN and SSS... have the same meaning as in an assembly listing line.

ERROR LINE

When an error is detected in a source image line, the line immediately following begins with

***ERROR

and contains one or more error codes beneath the portion of the source image line that is erroneous. Up to four error codes may be given for a single line. Table 3 lists the error codes and the severity level and significance of each code.

X ERROR IN SYMBOL

In most object languages, it is necessary for a processor to communicate values to the loader, in addition to generating object code to be placed in memory. Such values might be load origins, transfer addresses, or external values. It is a characteristic of the Sigma Standard Object Language that these values are not associated with physical record formats, but rather are passed as items of the language. A description of the Sigma Standard Object Language may be found in any of the XDS Monitor reference manuals.

The X error is generated when the symbol assembler is required to generate an object language expression containing a term which is not one of the following four types:

1. Integer
2. Address
3. External
4. Forward reference

The assembler generates an object language expression whenever it needs to communicate to the loader one of the following:

1. Load origin
2. Transfer address
3. DEF value
4. Nonstandard relocation
5. Satisfaction of forward reference

Example 42 illustrates several X type errors.

Example 42. X Type Assembly Listing Errors

	DEF	A	improper DEF value
	REF	B	
A	EQU	'ABCDEF'	
	GEN,16,16	C, B	
C	EQU	FS'1.5'	improper forward value
	END	'A'	improper transfer address; but notice that END B is permissible.

LITERAL LISTING LINE

Any literals evaluated during assembly are listed immediately following the END line or following the error line(s) containing the G and U codes, if they appear. Literals are listed in the order in which they were evaluated, and the listing line contains

LLLLL Current value of load location counter to word level in hexadecimal.

XXXXXXXX Value of literal as a hexadecimal memory word.

A Address classification flag.

Table 3. Symbol Error Codes

Code	Severity†	Significance
A	7	Arithmetic error caused when + or - operator is applied to items for which arithmetic operations are undefined (e.g., - address, L(value) + 1, BA(address - HA(address), etc.).
C	7	Constant string error caused when an explicit constant string contains an invalid character. This error also occurs when a decimal or floating-point constant occurs in a source line in a stand-alone environment and the BC specification has been given in the CMP control message (see XDS 90 10 53). The generated constant has an unpredictable value.
D	7	Duplicate definition error caused by an attempt to redefine a symbol that has already been defined in the symbol table. The first definition is retained and all subsequent definitions are flagged.
G	C	GOTO label has not been encountered prior to the END directive. All lines between the GOTO and END directives are ignored when the GOTO label is missing. In this case, the error line is printed after the END line.
I	7	Nonexistent instruction encountered in the command field of the symbolic line. A word of all zeros is generated for the instruction.
L	7	Label error caused by the absence of a label from a field that requires one. The position of the code indicates whether the flag applies to the label or argument field. This error code is also given for symbols having more than eight characters.
M	7	Mandatory field is not present.
O	3	Overflow (loss of significance) occurred during the conversion of a constant.
P	7	Parameter or usage error in a directive reference line (e.g., DEF, REF, or SREF out of order, no label on EQU, or noncharacter-constant for TEXT).
S	7	Syntax error encountered during expression evaluation (e.g., unpaired parentheses, illegal operation, etc.). A zero is returned for the expression value.
T	3	Truncation (loss of significance) occurred when a value was edited into a field.
U	7	Undefined symbol encountered in command or argument field of a source line (i.e., forward reference undefined at the end of local symbol region or at the end of the program). When this error occurs, the error line occurs after the END line, and it contains the error identifier, the error code, and the undefined symbol(s).
V	7	Invalid instruction error caused by a reference to an instruction that does not fall within the scope of the current SYSTEM directive (e.g., use of a floating-point instruction when only the basic instruction set was specified). The instruction is assembled correctly.
X	7	Object expression type is other than integer, address value, external reference, or forward reference. In this case, a zero value is generated for the invalid expression. For example, the source line END 'START' is invalid because 'START' is a character string constant and cannot be evaluated as the acceptable expression type for this directive.

†The highest severity code encountered during the assembly is passed to the loader as the second byte of the Module End load item for the object module. See BCM BP, RT Reference Manual, 90 09 53.

SYMBOL ABORT LINE

If an assembly requires more working space than is available in core memory, the assembler aborts the assembly and prints the message

SA

on the listing output. The assembler then reads the next source image line as the first line of a new assembly and continues to the next END directive.

ERROR COUNT LINE

If at least one error is encountered during an assembly, the last line of the listing output for that assembly contains the message

EC NNN

where NNN is the decimal number of source lines that contain errors.

SYMBOL DICTIONARY

At the end of each assembly listing, Symbol outputs a dictionary of all nonlocal symbols defined and/or referenced within the program. Local symbols are not included in the dictionary.

Nonlocal symbols are output in alphabetic order, sorted on the first four characters, and in ascending order of sequence: A-Z, 0-9, special characters.

The dictionary includes the following information arranged as illustrated:

```
Column    5.............19
          SYMBOL DICTIONARY
          SSSSSSS      DDDDDD
```

where

SSSSSSSS is the one- to eight-character symbol name.

DDDDDD is one of the following:

1. The value of the symbol in either address or constant format, according to its type.
2. X indicating an external reference (REF).
3. U indicating an undefined symbol.

SYMBOL CROSS-REFERENCE LISTING

An optional cross-reference (concordance) listing of all symbols used in the program can be produced along with the symbol dictionary. The cross-reference listing is produced by including the "CN" option on the Symbol Monitor control card. The format of this card and of the listing is explained in Chapter 7, "Symbol Operations".

7. OPERATIONS

Symbol has been designed to run under control of the Sigma Basic Control Monitor (BCM), Batch Processing Monitor (BPM), or Batch Time-Sharing Monitor (BTM). This chapter presents a brief discussion of Symbol operations under the BPM system. Assemblies under the BCM and BTM systems are processed in a similar manner, but the reader is advised to consult the appropriate Monitor reference manual for additional details.

ASSIGN CONTROL COMMAND

Appearing next in the run deck are any ASSIGN cards relating to the assembly. Normally, ASSIGN cards will not be needed, since the system has the following standard default assignments.

Logical Device or File	Physical Device
BO	Card punch
GO	Magnetic disk
LO	Line printer
SI	Card reader

BO, LO, and SI may be reassigned, by using the appropriate ASSIGN card.

SYMBOL CONTROL COMMAND

The Symbol control command has the following format:

```
! SYMBOL option ... , option
```

where the options are

BO specifies binary output.

CN specifies a concordance listing.

GO specifies an output GO file.

LO specifies listing output.

BA specifies batch-assembly mode.

The options may be specified in any order. If none are specified, BO and LO are assumed. Source input (SI) is always assumed.

PROGRAM DECK STRUCTURES

The Symbol assembler accepts only source images. If source input is from magnetic tape and the BA option has been specified, Symbol reads and assembles successive files until it encounters two successive end-of-file sentinels. If source input is from cards and the BA option has been specified, Symbol reads and assembles successive files until it encounters either two successive !EOD cards or any Monitor control card other than an !EOD card.

CONCORDANCE LISTING

A concordance (cross-reference) listing of all symbols used in the program is produced when the CN option is given on the Symbol control command. The listing is produced by modifying the standard symbol dictionary listing, which prints for every assembly.

The information printed is symbol name, value, and reference line numbers. A sample entry might appear as

```
ALPHA    00000005    10    17    22*    30
```

which means that symbol ALPHA has the hexadecimal value 5 and appears on source program line numbers 10, 17, 22, and 30. The asterisk following line 22 means that ALPHA appeared in the label field.

Concordance information is memory resident and increases space requirements by one word per reference.

BTM OPERATIONS

Input is typed directly at the user's terminal or from a file. Output is a program listing and/or an assembled object program which may be loaded and executed by the Loader subsystem (see BTM User's Guide, 90 16 79).

INPUT/OUTPUT ASSIGNMENTS

Prior to calling the Symbol subsystem, it is possible to make input/output assignments by use of the Executive ASSIGN command. Input/output assignments are listed in Table 4.

Table 4. Input/Output Assignments

Symbol	Description
M:SI	Source language input. The default assignment is to the user's terminal. An alternative is for the user to specify a file previously created by use of the EDIT subsystem. Listing output. Default assignment is to the user's terminal.

Table 4. Input/Output Assignments (cont.)

Symbol	Description
M:BO	Binary output of assembled object program. By default this goes into temporary file BOTEMPx, where "x" is the special ID for the user's terminal. The user may also specify a file of his own. This is the file to be specified to the Loader when it is desired to run the program.

ASSEMBLER OPTIONS

The subsystem is called following the Executive prompt character by typing SY. The Executive will then type the rest of the word and turn control over to the Symbol Subsystem, which then requests a list of options. The operator may specify options listed in Table 5, separating them with commas. If no options are specified (carriage return only), all the options listed are assumed. If the operator specifies any options, he gets only those options.

Table 5. Symbol Options

Symbol	Option
BO	Binary output of an assembled object program.
LO	Output a program listing.
CN	Include a cross-reference list in the program listing. This must be used in conjunction with the LO option; CN is meaningless if used alone.
SD	Include special symbol tables for use by the Loader subsystem's debugging feature at run-time.

The following is an example of a Symbol assembly with source input from a file on the disk, and listing output to a file on the disk. All options are selected with the exception of the debugging feature symbol tables (SD).

```
!ASSIGN M:LO, (FILE, CMPLO)

!ASSIGN M:SI, (FILE, CMPS)

!SYMBOL

OPTIONS: BO, LO, CN

* * END OF ASSEMBLY * *
```

LISTING FORMAT

If the program listing is typed on the user's terminal, it will automatically be reformatted to fit the carriage width. Each listing line will be typed as two lines:

1. The first line will contain the source image.
2. The second line will contain the line number and object code portion of the normal listing. In addition, if the source file was on disk in EDIT format, the EDIT file sequence number will be typed in decimal format.

If the assembly listing is not displayed at the terminal, any errors found in the assembly are displayed both at the terminal and in the listing file. Three lines are typed at the terminal:

1. The offending source line.
2. The normal Symbol error indicator (*****) and a letter positioned under the image.
3. The line number, object code produced, and sequence number of the record.

APPENDIX A: SUMMARY OF SYMBOL DIRECTIVES

In this summary brackets are used to indicate optional items.

Form			Function	Page
[label]	ASECT		Declares program section "label" as an absolute section with no memory protection and sets location counters to absolute zero.	19
	BOUND	boundary	Advances the execution location counter to a byte multiple of "boundary" and advances the load location counter the same number of bytes.	16
name	COM, field list	value list	Describes a command skeleton; "v_i" specifies the contents of each "field"; "label" is the symbol by which the command skeleton is referenced.	27
[label]	CSECT	[value]	Declares program section "label" as a relocatable control section.	19
[label]	DATA[, f]	$value_1$[, . . . , $value_n$]	Generates each value in the list of v_i into a field whose size is specified by f in bytes. If f is omitted, a field size of 4 bytes is assumed.	28
	DEF	$symbol_1$ [, . . . , $symbol_n$]	Declares that the "$symbol_i$" may be referenced by other separately assembled programs.	25
[label]	DO1	exp	If the value of "exp" is greater than zero, processes the one statement following the DO1, "exp" times, then continues the assembly at the next statement. If "exp" ≤ 0, skips the statement following DO1 and resumes the assembly.	22
	END	[exp]	Terminates the program. Optionally provides the starting address of the program.	22
label	EQU	exp	Sets "label" equal to the value of "exp".	24
[label]	GEN, field list	value list	Produces a hexadecimal value representing v_i in the number of bits specified by each field in "field list".	26
	GOTO[, k]	$label_1$[, . . . , $label_n$]	Resumes assembly at the statement whose label corresponds to the kth "label".	22
[label]	LOC [, n]	location	Sets the execution location counter (S) to the value "location" and sets its resolution specification to n, where the value of n is 1, 2, 4, or 8.	16
	LOCAL	[$name_1$, . . . , $name_n$]	Terminates existing local symbol region and initiates a new region where the "$name_i$" are local symbols.	23
label	ORG [, n]	location	Sets both the load location counter (SS) and the execution location counter (S) to the value "location" and sets their resolution specifications to n, where the value of n is 1, 2, 4, or 8.	16
	PAGE		Upspaces assembly listing to the top of form.	30
	REF	$symbol_1$ [, . . . , $symbol_n$]	Declares that the "$symbol_i$" are references to externally defined symbols.	25

Form			Function	Page
[label]	RES[,n]	u	Advances both location counters ($ and $$) by u n-sized units. If n is omitted, a size of 4 bytes is assumed.	18
	SREF	$symbol_1[,\ldots,symbol_n]$	Declares that the "$symbol_i$" are secondary external references.	25
	SYSTEM	name	Indicates which instruction set is correct for the assembly.	21
[label]	TEXT	'cs'	Assembles "cs" (character string constant) in binary-coded format for use as an output message.	30
[label]	TEXTC	'cs'	Assembles "cs" (character string constant) in binary-coded format, preceded by a byte count, for use as an output message.	30

APPENDIX B. SUMMARY OF SIGMA INSTRUCTION MNEMONICS

Required syntax items are underlined whereas optional items are not. The following abbreviations are used:

m	mnemonic
r	register expression
v	value expression
*	indirect designator
a	address expression
x	index expression
d	displacement expression

Codes for required options are

7	Sigma 6 or 7
P	Privileged
D	Decimal Option
F	Floating-Point Option
L	Lock Option
MP	Memory Map Option
SF	Special Feature – not implemented on all machines

Mnemonic	Syntax	Function	Equivalent To:	Required Options
LOAD/STORE				
LI	m,r v	Load Immediate		
LB	m,r *a,x	Load Byte		
LH	m,r *a,x	Load Halfword		
LW	m,r *a,x	Load Word		
LD	m,r *a,x	Load Doubleword		
LCH	m,r *a,x	Load Complement Halfword		
LAH	m,r *a,x	Load Absolute Halfword		
LCW	m,r *a,x	Load Complement Word		
LAW	m,r *a,x	Load Absolute Word		
LCD	m,r *a,x	Load Complement Doubleword		
LAD	m,r *a,x	Load Absolute Doubleword		
LS	m,r *a,x	Load Selective		
LM	m,r *a,x	Load Multiple		
LCFI	m v,v	Load Conditions and Floating Control Immediate		
LCI	m v	Load Conditions Immediate		
LFI	m v	Load Floating Control Immediate		
LC	m *a,x	Load Conditions		
LF	m *a,x	Load Floating Control		
LCF	m *a,x	Load Conditions and Floating Control		
LAS	m,r *a,x	Load and Set		SF
LMS	m,r *a,x	Load Memory Status		SF
XW	m,r *a,x	Exchange Word		
STB	m,r *a,x	Store Byte		
STH	m,r *a,x	Store Halfword		
STW	m,r *a,x	Store Word		
STD	m,r *a,x	Store Doubleword		
STS	m,r *a,x	Store Selective		
STM	m,r *a,x	Store Multiple		
STCF	m *a,x	Store Conditions and Floating Control		
ANALYZE AND INTERPRET				
ANLZ	m,r *a,x	Analyze		
INT	m,r *a,x	Interpret		
FIXED-POINT ARITHMETIC				
AI	m,r v	Add Immediate		
AH	m,r *a,r	Add Halfword		
AW	m,r *a,x	Add Word		
AD	m,r *a,x	Add Doubleword		
SH	m,r *a,x	Subtract Halfword		

Mnemonic	Syntax	Function	Equivalent To:	Required Options
FIXED-POINT ARITHMETIC (cont.)				
SW	m, r *a, x	Subtract Word		
SD	m, r *a, x	Subtract Doubleword		
MI	m, r v	Multiply Immediate		
MH	m, v *a, x	Multiply Halfword		
MW	m, r *a, x	Multiply Word		
DH	m, r *a, x	Divide Halfword		
DW	m, r *a, x	Divide Word		
AWM	m, r *a, x	Add Word to Memory		
MTB	m, v *a, x	Modify and Test Byte		
MTH	m, v *a, x	Modify and Test Halfword		
MTW	m, v *a, x	Modify and Test Word		
COMPARISON				
CI	m, r v	Compare Immediate		
CB	m, r *a, x	Compare Byte		
CH	m, r *a, x	Compare Halfword		
CW	m, r *a, x	Compare Word		
CD	m, r *a, x	Compare Doubleword		
CS	m, r *a, x	Compare Selective		
CLR	m, r *a, x	Compare with Limits in Register		
CLM	m, r *a, x	Compare with Limits in Memory		
LOGICAL				
OR	m, r *a, x	OR Word		
EOR	m, r *a, x	Exclusive OR Word		
AND	m, r *a, x	AND Word		
SHIFT				
S	m, r *a, x	Shift		
SLS	m, r v, x	Shift Logical, Single		
SLD	m, r v, x	Shift Logical, Double		
SCS	m, r v, x	Shift Circular, Single		
SCD	m, r v, x	Shift Circular, Double		
SAS	m, r v, x	Shift Arithmetic, Single		
SAD	m, r v, x	Shift Arithmetic, Double		
SF	m, r *a, x	Shift Floating		
SFS	m, r v, x	Shift Floating, Short		
SFL	m, r v, x	Shift Floating, Long		
CONVERSION				
CVA	m, r *a, x	Convert by Addition		7
CVS	m, r *a, x	Convert by Subtraction		7
FLOATING-POINT ARITHMETIC				
FAS	m, r *a, x	Floating Add Short		F
FAL	m, r *a, x	Floating Add Long		F
FSS	m, r *a, x	Floating Subtract Short		F
FSL	m, r *a, x	Floating Subtract Long		F
FMS	m, r *a, x	Floating Multiply Short		F
FML	m, r *a, x	Floating Multiply Long		F
FDS	m, r *a, x	Floating Divide Short		F
FDL	m, r *a, x	Floating Divide Long		F

Mnemonic	Syntax		Function	Equivalent To:		Required Options
DECIMAL	(Decimal instructions are standard on Sigma 6.)					
DL	m, v	*a, x	Decimal Load			D
DST	m, v	*a, x	Decimal Store			D
DA	m, v	*a, x	Decimal Add			D
DS	m, v	*a, x	Decimal Subtract			D
DM	m, v	*a, x	Decimal Multiply			D
DD	m, v	*a, x	Decimal Divide			D
DC	m, v	*a, x	Decimal Compare			D
DSA	m	*a, x	Decimal Shift Arithmetic			D
PACK	m, v	*a, x	Pack Decimal Digits			D
UNPK	m, v	*a, x	Unpack Decimal Digits			
BYTE STRING						
MBS	m, r	d	Move Byte String			7
CBS	m, r	d	Compare Byte String			7
TBS	m, r	d	Translate Byte String			7
TTBS	m, r	d	Translate and Test Byte String			7
EBS	m, r	d	Edit Byte String			D
PUSH DOWN						
PSW	m, r	*a, x	Push Word			
PLW	m, r	*a, x	Pull Word			
PSM	m, r	*a, x	Push Multiple			
PLM	m, r	*a, x	Pull Multiple			
MSP	m, r	*a, x	Modify Stack Pointer			
EXECUTE/BRANCH						
EXU	m	*a, x	Execute			
BCS	m, v	*a, x	Branch on Conditions Set			
BCR	m, v	*a, x	Branch on Conditions Reset			
BIR	m, r	*a, x	Branch on Incrementing Register			
BDR	m, r	*a, x	Branch on Decrementing Register			
BAL	m, r	*a, x	Branch and Link			
B	m	*a, x	Branch	BCR, 0	*a, x	
BE	m	*a, x	Branch if Equal	BCR, 3	*a, x	
BG	m	*a, x	Branch if Greater Than	BCS, 2	*a, x	
BGE	m	*a, x	Branch if Greater Than or Equal to	BCR, 1	*a, x	
BL	m	*a, x	Branch if Less Than	BCS, 1	*a, x	
BLE	m	*a, x	Branch if Less Than or Equal to	BCR, 2	*a, x	
BNE	m	*a, x	Branch if Not Equal	BCS, 3	*a, x	
BAZ	m	*a, x	Branch if Implicit AND is Zero[t]	BCR, 4	*a, x	
BANZ	m	*a, x	Branch if Implicit AND is Nonzero[t]	BCS, 4	*a, x	
BEZ	m	*a, x	Branch if Equal to Zero	BCR, 3	*a, x	
BNEZ	m	*a, x	Branch if Not Equal to Zero	BCS, 3	*a, x	
BGZ	m	*a, x	Branch if Greater Than Zero	BCS, 2	*a, x	
BGEZ	m	*a, x	Branch if Greater Than or Equal to Zero	BCR, 1	*a, x	
BLZ	m	*a, x	Branch if Less Than Zero	BCS, 1	*a, x	
BLEZ	m	*a, x	Branch if Less Than or Equal to Zero	BCR, 2	*a, x	

For Use After Comparison Instructions

[t] See CW instruction in Xerox Sigma Computer Reference Manual.

Mnemonic	Syntax		Function		Equivalent To:		Required Options
EXECUTE/BRANCH (cont.)							
BOV	m	*a, x	For Use After Fixed-Point Arithmetic Instructions	Branch if Overflow	BCS, 4	*a, x	
BNOV	m	*a, x		Branch if No Overflow	BCR, 4	*a, x	
BC	m	*a, x		Branch if Carry	BCS, 8	*a, x	
BNC	m	*a, x		Branch if No Carry	BCR, 8	*a, x	
BNCNO	m	*a, x		Branch if No Carry and No Overflow	BCR, 12	*a, x	
BWP	m	*a, x		Branch if Word Product	BCR, 4	*a, x	
BDP	m	*a, x		Branch if Doubleword Product	BCS, 4	*a, x	
BEV	m	*a, x	For Use After Fixed-Point Shift Instructions	Branch if Even (number of 1's shifted)	BCR, 8	*a, x	
BOD	m	*a, x		Branch if Odd (number of 1's shifted)	BCS, 8	*a, x	
BID	m	*a, x	For Use After Decimal Instructions	Branch if Illegal Decimal Dig	BCS, 8	*a, x	
BLD	m	*a, x		Branch if Legal Decimal Digit	BCR, 8	*a, x	
BSU	m	*a, x	For Use After Push Down Instructions	Branch if Stack Underflow	BCS, 2	*a, x	
BNSU	m	*a, x		Branch if No Stack Underflow	BCR, 10	*a, x	
BSE	m	*a, x		Branch if Stack Empty	BCS, 1	*a, x	
BSNE	m	*a, x		Branch if Stack Not Empty	BCR, 1	*a, x	
BSF	m	*a, x		Branch if Stack Full	BCS, 4	*a, x	
BSNF	m	*a, x		Branch if Stack Not Full	BCR, 15	*a, x	
BSO	m	*a, x		Branch if Stack Overflow	BCS, 8	*a, x	
BNSO	m	*a, x		Branch if No Stack Overflow	BCR, 8	*a, x	
BIOAR	m	*a, x	For Use After Input/Output Instructions	Branch if I/O Address Recognized	BCR, 8	*a, x	
BIOANR	m	*a, x		Branch if I/O Address Not Recognized	BCS, 8	*a, x	
BIODO	m	*a, x		Branch if I/O Device Operating	BCS, 4	*a, x	
BIODNO	m	*a, x		Branch if I/O Device Not Operating	BCR, 4	*a, x	
BIOSP	m	*a, x		Branch if I/O Start Possible	BCR, 4	*a, x	
BIOSNP	m	*a, x		Branch if I/O Start Not Possible	BCS, 4	*a, x	
BIOSS	m	*a, x		Branch if I/O Start Successful	BCR, 4	*a, x	
BIOSNS	m	*a, x		Branch if I/O Start Not Successful	BCS, 4	*a, x	
CALL							
CAL1	m, v	*a, x		Call 1			
CAL2	m, v	*a, x		Call 2			
CAL3	m, v	*a, x		Call 3			
CAL4	m, v	*a, x		Call 4			
CONTROL							
LPSD	m, r	*a, x		Load Program Status Doubleword			P
XPSD	m, r	*a, x		Exchange Program Status Doubleword			P
LRP	m	*a, x		Load Register Pointer			P
MMC	m, r	v		Move to Memory Control			P
LMAP	m, r			Load Map			7MP
LPC	m, r			Load Program Control			7MP
LLOCKS	m, r			Load Locks			LP
WAIT	m	*a, x		Wait			P
RD	m, r	*a, x		Read Direct			P
WD	m, r	*a, x		Write Direct			P
NOP[t]	m	*a, x		No Operation			
PZE	m	*a, x		Positive Zero			

[t] Equivalent to a BCS instruction with r = 0.

Mnemonic	Syntax	Function	Equivalent To:	Required Options
INPUT/OUTPUT				
SIO	m,r *a,x	Start Input/Output		P
HIO	m,r *a,x	Halt Input/Output		P
TIO	m,r *a,x	Test Input/Output		P
TDV	m,r *a,x	Test Device		P
AIO	m,r *a,x	Acknowledge Input/Output Interrupt		P

XEROX

Reader Comment Form

We would appreciate your comments and suggestions for improving this publication.

Publication No.	Rev. Letter	Title	Current Date

How did you use this publication?

☐ Learning ☐ Installing ☐ Sales
☐ Reference ☐ Maintaining ☐ Operating

Is the material presented effectively?

☐ Fully Covered ☐ Well Illustrated ☐ Well Organized ☐ Clear

What is your overall rating of this publication?

☐ Very Good ☐ Fair ☐ Very Poor
☐ Good ☐ Poor

What is your occupation?

Your other comments may be entered here. Please be specific and give page, column, and line number references where applicable. To report errors, Please use the Xerox Software Improvement or Difficulty Report (1188) instead of this form.

Your Name & Return Address

2190(12 72)

Thank You For Your Interest. (fold & fasten as shown on back, no postage needed if mailed in U.S.A.)

CUT ALONG

FOLD ALONG LINE

PLEASE FOLD AND TAPE –
NOTE: U. S. Postal Service will not deliver stapled forms

FIRST CLASS
PERMIT NO. 39531
WALTHAM, MA
02154

Business Reply Mail
Postage Stamp Not Necessary if Mailed in the United States

Postage Will Be Paid By:

HONEYWELL INFORMATION SYSTEMS
200 SMITH STREET
WALTHAM, MA 02154

ATTENTION: PUBLICATIONS, MS 486

FOLD ALONG LINE

Honeywell

Honeywell Information Systems
In the U.S.A.: 200 Smith Street, MS 486, Waltham, Massachusetts 02154
In Canada: 2025 Sheppard Avenue East, Willowdale, Ontario M2J 1W5
In Mexico: Avenida Nuevo Leon 250, Mexico 11, D.F.

16514, 3C976, Printed in U.S.A.

XM06, Rev. 0