15090D

SOLOMON PROJECT TECHNICAL
MEMORANDUM NO. 24

SOLOMON II Assembly System

4 November 1963

WESTINGHOUSE DEFENSE AND SPACE CENTER
Defense and Space Systems Operations
Baltimore, Maryland

TABLE OF CONTENTS

1. INTRODUCTION AND DEFINITIONS

2. THE CODING FORM

3. ADDRESSES

LIST OF ILLUSTRATIONS

1. INTRODUCTION AND DEFINITIONS

1.1 INTRODUCTION

This manual is designed to teach the SOLOMON assembly language to programmers already familiar with the SOLOMON system. No attempt is made to teach anything about SOLOMON; for this purpose, see "SOLOMON II Programmer's Reference Manual."

Unfortunately, there is no simple way of describing a language. In many cases, it has been necessary in this manual to mention things that are not yet defined. On the other hand, it is hoped that the arrangement of the manual will make it suitable for use as a reference manual as well.

1.2 DEFINITIONS

To facilitate full understanding of this manual, two basic definitions are given at the outset.

1.2.1 Location Counter

In order to assign the proper value to each address of a memory location, the assembler uses a special cell called the Location Counter (see SEG in Pseudo-Instructions). The location counter can be initially set to an arbitrary value. The first line of the program is assigned to this memory location. For each machine instruction processed, the Location Counter is increased by 1 normally. Certain instructions, though, may result in an increase of more than 1.

1.2.2 Sequence Break

Any condition, instruction, or pseudo-instruction which causes the Location Counter to increment by any other number than +1 is called a sequence break, i.e., a break in the normal sequential pattern of assigning lines of a program to memory locations.

2. THE CODING FORM

2.1 GENERAL

This section is to familiarize the reader with the SOLOMON Coding Form; detailed discussions concerning the various fields of this form are presented in subsequent sections.

A sample coding form for the assembly language is shown on the following page. Each line on this coding form represents a single punched card and in general represents a single instruction. Sometimes, however, an instruction may require more than one card, or a card may contain something other than an instruction.

A field on a punched card is defined as a (specified) number of adjacent columns (special case - a single column) in which are punched a given type of data. The name of a field is associated with all of the columns of the field. The card is divided into six fields. The following discussions of the fields pertain primarily to machine instructions. Pseudo-instructions and others may require different usage.

2.2 THE T FIELD

("T" for "TYPE") specifies the type of card and gives the assembler a general indication of the kind of processing that will be required. Five basic entries may be made in the T field.

a. The field may be left blank.* This specifies that normal assembly processing is to take place.

---

* Many symbols have been used in the literature to symbolize a blank column. In this manual, the symbol ␣ will be used, primarily for its international origin. Other symbols include: #(American typography), Δ(Remington Rand), and ⌀ (IBM).

b. The field may be punched with the letter C. This specifies that the card contains a comment which is to be reproduced in the assembly listing; it has no other effect on the assembler.

c. The field may be punched with an asterisk (*). This card is also treated as a comment, but the line will be the first line of a new page of the assembly listing. (This facility allows the programmer to have some sort of editing of his listings - a useful aid when the assembly listing is to be included as a report.)

d. The field may be numbered with a number from 1 to 9. This specifies that the card is to be considered as a continuation of the previous card.

e. The field may contain the letter X. This indicates that the card is to be completely ignored by the assembler and is not to appear in the output listing. This type of field will usually be inserted by the assembler into those lines which represent one time pseudo-operations such as LON, TRACE, and UNTRACE (q. v.).

Certain errors are detected by the assembler. The errors which may be made in the T field are fairly few and will be listed here.

a. If an instruction requires more than one card, the first continuation card must be numbered 1 in the T field, the second, 2, etc.

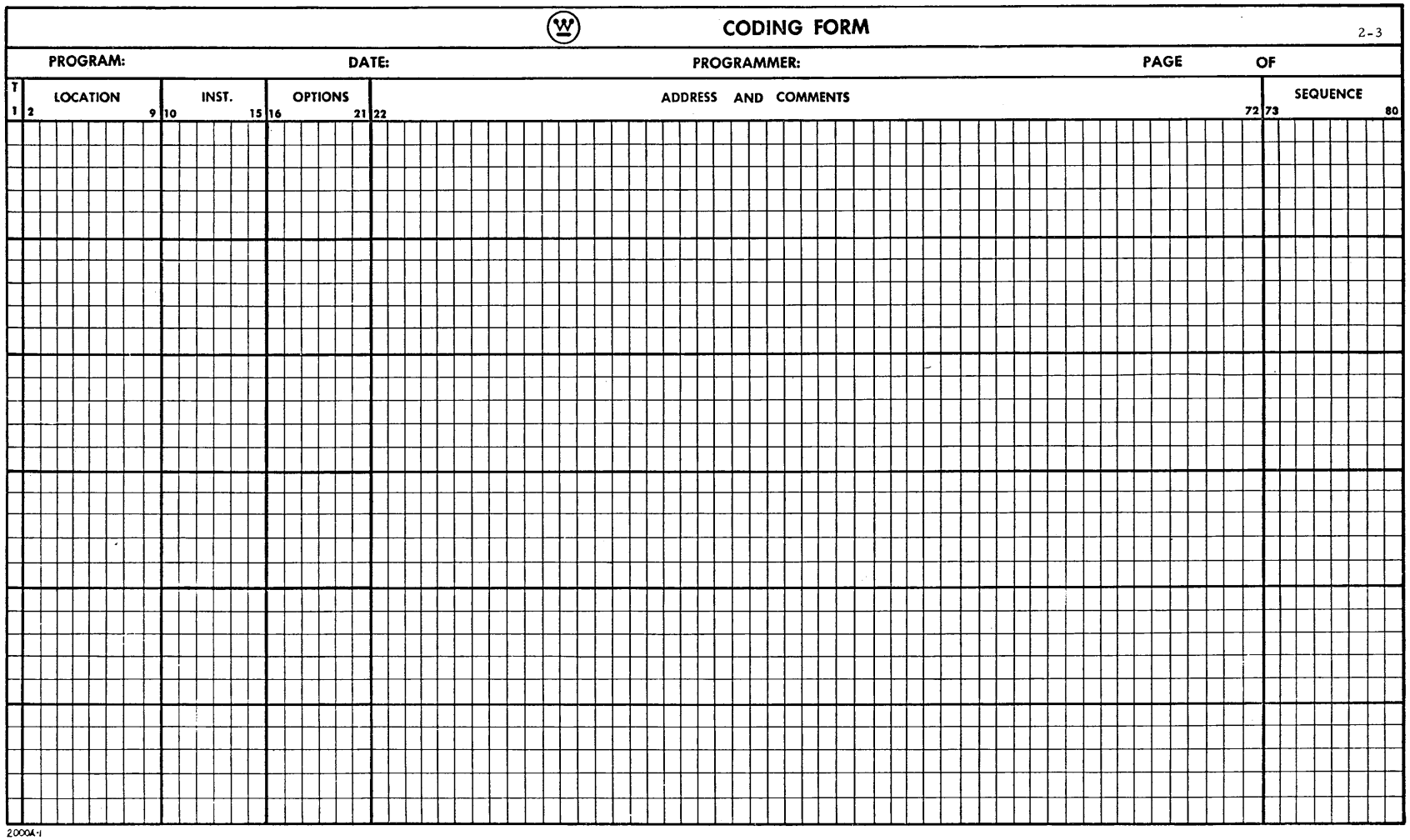
b. If the T field is numeric, the LOCATION, INST, and OPTIONS Fields must be blank.

c. Only the characters specified in this section (␣, C, *, 1, 2, 3, 4, 5, 6, 7, 8, 9 and X) may appear in the T field.

Errors are indicated on the assembly listing. In addition to indicating the errors in the T field, the assembler will treat the erroneous card (and all following cards until the next one with the T field blank) as though they were comment cards. As a consequence, an error in the T field may generate additional errors in the remainder of the program.

2.3 THE LOCATION FIELD

The LOCATION Field contains an indication of the location of the line of coding specified by the card. (Obviously, if the card does not specify a line


CODING FORM
2-3
PROGRAM:
DATE:
PROGRAMMER:
PAGE
OF
T
1
LOCATION
2
9
INST.
10
15
OPTIONS
16
21
ADDRESS AND COMMENTS
22
72
SEQUENCE
73
80
2000A-1

of coding, the LOCATION Field may have other uses.) In general, the LOCATION Field is blank, which indicates that the line of coding is to be considered as the immediate successor of the preceding line of coding. Other types of indications may be written in the LOCATION Field; they are discussed in the section on Addresses.

2.4 THE INSTRUCTION FIELD

The INST Field contains, in general, mnemonics for the machine operations to be performed.

2.5 THE OPTIONS FIELD

The OPTIONS Field is used to specify values to appear in the IA, MD, MV, and G fields of a machine instruction. Symbols which may be used are:

a. * - Indicates indirect addressing (IA = 1).

b. 0123 - Indicates modes. If an instruction requires both an MD and MV value, the rightmost digit specifies MV. All other digits specify MD. If no digits appear and an MD value is required, all modes are assumed. If an MV value is required and no digits appear, zero is assumed.

c. A - Indicates the use of all modes in MD. This is equivalent to 0123 or no digits at all.

d. N - Indicates no modes for MD (MD = 0).

e. R - Indicates that row geometric control is to be used.

f. C - Indicates that column geometric control is to be used.

g. G - Indicates that both row and column geometric control are to be used. This is equivalent to writing RC.

h. Boolean Mnemonic - Indicates the Boolean operation which is to appear in the MD field. (See "SOLOMON II Programmers Reference Manual" for mnemonics and relevant instructions.)

Except for the MV value relative to MD values and the Boolean mnemonics, the order of characters in the OPTIONS Field is not significant.

Examples:

a. Assume a mode setting instruction.

OPTIONS						ADDRESSES AND COMMENTS
*	0	1		3	R	$ 1
1	*	0	R		3	$ 2
		G		0		$ 3
						$ 4

(1) Indirect addressing; use only PE's in modes 0 or 1; set to mode 3; and use row geometric control.

(2) Equivalent to 1.

(3) Use all PE's; set to mode 0; and use both row and column geometric control

(4) Use all PE's and set to mode 0.

b. Assume a nonmode setting instruction.

OPTIONS						ADDRESSES AND COMMENTS
*	0	1		3	R	$ 5
W	A	M		*		$ 6
*	W	A	M			$ 7
						$ 8

(5) Indirect addressing; use only PE's in modes 0, 1, or 3; and use row geometric control.

(6) Indirect addressing; perform the Boolean operation WAM (only with a valid instruction).

(7) Equivalent to 6.

(8) Use all modes if MD setting is relevant.

2.6 THE ADDRESSES AND COMMENTS FIELDS

This field is used to specify values to appear in the R, M, and X fields of a machine instruction. It may also contain comments which are printed on the listing but do not otherwise affect the program.

The general format is: route, base adr, index, $ comments

route - Indicates the route, when relevant, by one of the route mnemonics: I, N, E, S, W, B, R, C, or a number 0 through 7. If route is blank and a route value is required, zero is used.

base adr - Indicates the value to appear in the M field of the instruction. Any of the forms described in Section 3 may be used.

index - Indicates the index register to be used. An absolute or symbolic address form (see Section 3) may be used. Address arithmetic may be used.

If either base adr or index is blank, a value of zero is used. Commas may be omitted if the field is blank between them and the dollar sign. Unless the instruction requires a route value and the leftmost parameter is an acceptable route form, the leftmost parameter is base adr. All PE instructions are considered to require route specifications in this sense.

Examples:

a. Assume a PE instruction.

ADDRESSES AND COMMENTS	
N	$ R FIELD = N, M FIELD = 0, X FIELD = 0
, N	$ R FIELD = 0, M FIELD = N, X FIELD = 0
SAM	$ R FIELD = 0, M FIELD = SAM, X FIELD = 0
, SAM	$ R FIELD = 0, M FIELD = SAM, X FIELD = 0
N, A, 2	$ R FIELD = N, M FIELD = A, X FIELD = 2

b. Assume an NCU instruction.

ADDRESSES AND COMMENTS	
N	M FIELD = N, X FIELD = 0
, N	M FIELD = 0, X FIELD = N
N, A	M FIELD = N, X FIELD = A

2.7 THE SEQUENCE FIELD

The final field on the card is the SEQUENCE Field. This field may be used by the programmer to identify his cards in any desired manner. It is reproduced on the assembly listing but has no other effect.

3. ADDRESSES

3.1 GENERAL

An address, as used in this manual, is any way of referring in the assembly language to a memory location in the computer. It may also refer to other types of parameters (e.g., number of places to shift).

There are five types of addresses which may be used. They are:

a. Absolute

b. Symbolic

c. Regional

d. Program Point

e. Pool Constant

These are discussed in the following sections and summarized in table 3-1.

3.2 TYPES OF ADDRESSES

This section lists the types of addresses in table 3-1 and discusses them on subsequent pages.

TABLE 3-1

TYPES OF ADDRESSES

Type of Addresses	LOCATION Field	ADDRESS Field
Blank	This line is the physical successor of previous line	Field not significant, will assemble as zeros.
Program Point	Defines the program point	Suffix F - forward prog. pt. reference B - backward prog. pt. reference
Absolute	Specific location of the line	Assemble to specific absolute address.

TABLE 3-1 (Continued)

Type of Addresses	LOCATION Field	ADDRESS Field
Undefined Symbolic	Defines Symbolic	Assembles as 0's if never defined - error indication Defined Address - if ultimately defined
Defined Symbolic	Error Indications Line = Comment	Assembles as defined address
Undefined Regional	Error Indication Line = Comment	Assembles as defined address, if ultimately defined or error indication 0's, if never defined.
Defined Regional	First occurrence-location of line does not affect program counter. Other occurrences - Line = Comment	Assembles as defined address
Constant	Error Indication Line = Comment	Assembles as the address of the memory location assigned by the assembler to the specified constant.

3.2.1 Absolute Addresses

An absolute address is a decimal or octal number of not more than six digits.* This number will be treated modulo the memory size of the computer. If an absolute address (decimal only) appears in the LOCATION Field and is not less than the Location Counter, the line is assigned to the specified absolute address and following lines are assigned to conventional increasing addresses. If appropriate, a sequence break is indicated on the listing. If the absolute address is less than the Location Counter, an error is indicated

* In the LOCATION Field, at least two digits must be written to distinguish absolute addresses from program points.

and the LOCATION Field is ignored. An absolute address in the ADDRESS Field assembles as the specified absolute address.

Octal absolute addresses are distinguished from decimal absolute addresses by being preceded by M/ (e.g., M/34567).

3.2.2 Symbolic Addresses

A symbolic address is any combination of characters either alphabetic or numeric which is not of a format reserved for regional, absolute, or program point addresses and, of course, which does not exceed the field size.

a. Undefined Symbolic - an undefined symbolic address is a symbolic address which, when encountered, has not appeared in the LOCATION Field of the same or a previous line of coding.

The occurrence of an undefined symbolic address in the LOCATION Field defines it (i.e., assigns the address into which the line of coding is placed as the value of the symbol).

If an undefined symbolic address appears in the ADDRESS Field, two possible results may follow. If the symbol is later defined, (i.e., appears in the LOCATION Field of some later line), then the symbol will assemble as the later defined address. If the symbol is never defined, it will assemble as zeros and an error indication is given (but see the DEF pseudo-op).

b. Defined Symbolic Address - When a symbolic address which has been defined (has appeared in the LOCATION Field of some line) is again encountered in the LOCATION Field of some later line, then the later line is treated as a comment line and an error indication is given. This is an attempt to assign two different lines to the same location. The error line will appear in the listing (but see the NAME pseudo-op).

Obviously, a defined symbolic address in the ADDRESS Field is a normal thing to find and will assemble as the defined address.

3.2.3 Regional Addresses

A regional address is a relative address written in terms of the base (first line) in a region (block of memory). Unlike a symbolic or a program point address, a region is defined by a pseudo-instruction (see REG, paragraph

5.2) and not by its occurrence in a LOCATION Field. It is symbolized by one, two, or three alphabetic characters followed by at least three numeric characters (ABC1234, F234, MG34567).

a. Defined Regional - if a defined regional address occurs in the LOCATION Field, the line is assigned to the regional address. Successive lines are assembled into conventional increasing addresses in the same manner as when an absolute address is encountered. The same type of errors are indicated as are described in Section 3.

A defined regional in the ADDRESS Field assembles the regional address as the corresponding absolute address.

b. Undefined Regional - an undefined regional in the LOCATION Field results in an error indication and the line is treated as a comment (the line appears in the listing but does not affect the program).

An undefined regional, when encountered in the ADDRESS Field, can result in two things. If the undefined regional is defined at some later point in the program, the symbol will assemble as the defined address. If the undefined regional is never defined, an error indication is given and the regional address will assemble as zeros. (The DEF pseudo-op has no effect in this case.)

3.2.4 Program Point Addresses

A program point is a redefinable reference point for lines of coding which are (usually) close to each other in the program. A program point is written as a single decimal digit and its occurrence in the LOCATION Field defines the program point.

When a program point occurs in the ADDRESSES AND COMMENTS Field it is written as a single decimal digit, suffixed by either an F or a B. In other words, a program point in the ADDRESS Field will have the general form: 6F, 5B, 5F, etc. If the suffix is F, this means reference is made forward in the program to the next line having the same number in the LOCATION Field as that number appearing before F. The suffix B is the same except reference is made backward in the program.

The diagram in figure 3-1 illustrates program point referencing.

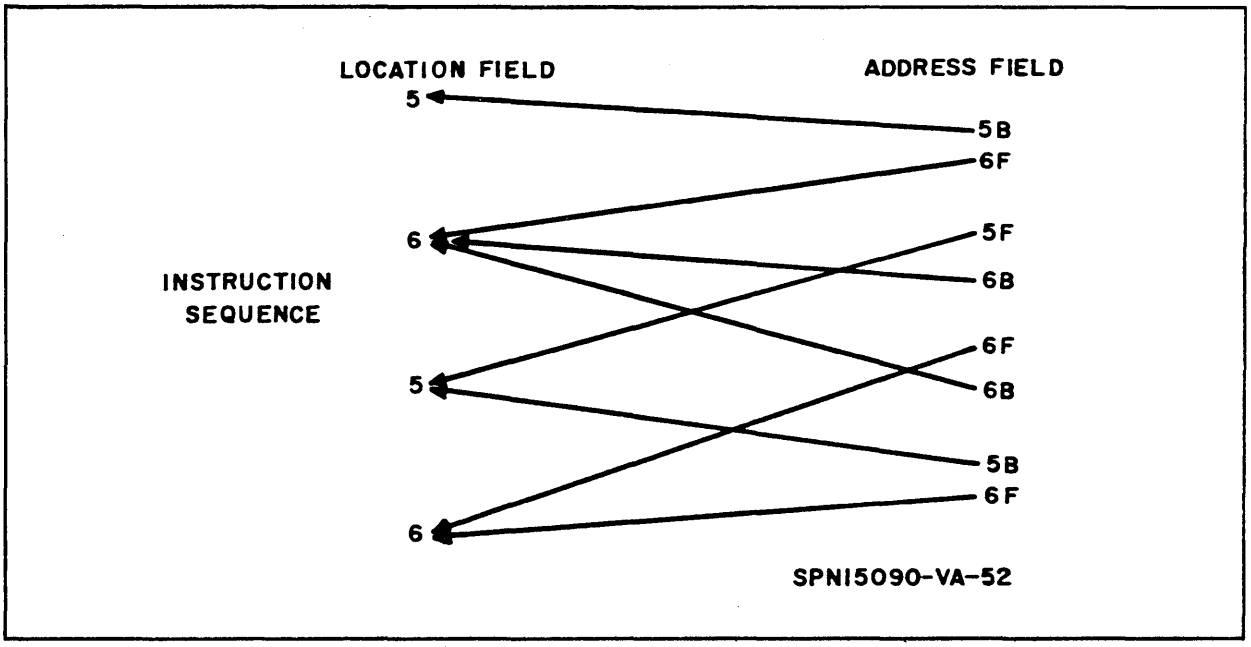


Figure 3-1. Program Point Referencing

It should be clear that any errors made in program point referencing in general will not be detected. If the programmer neglects to define a program point and then refers to it, the reference will be made to the next line which has the same program point. For example, in figure 3-1, if the second 5 did not appear in the LOCATION Field, then all of the 5 B's in the ADDRESS Field would refer to the first 5 in the LOCATION Field and no error would be detected.

3.2.5 Pool Constant Addresses

Constants may be written as pool constants. A pool constant is stored in a separate section of memory and may be used repeatedly in a program merely by accessing the memory location at which the constant is stored. Refer to Section 4 for a detailed description of pool constants.

3.2.6 Address Arithmetic

Arithmetic expressions involving the operations addition, subtraction, multiplication, and division are permitted in the ADDRESS AND COMMENTS Field. Any of the basic address forms except pool constants may be used in

these expressions. Multiplication and division (indicated by * and :, respectively) are done before addition and subtraction. Operations are performed in left-to-right order with any remainders being dropped and with each result modulo the memory size.

The symbol * may be interpreted as either an operation (multiplication) or an operand (the value of the Location Counter) depending on the context.

Example 1:

```
**2
```

The current location times two.

Example 2:

SAM+**2:JØE-M/67

The current location is multiplied by 2 and divided by the value of JØE. This result is added to the value of SAM, and $67_{(8)}$ is subtracted from the sum.

4. POOL CONSTANTS

4.1 GENERAL

The constant pool will contain a given bit configuration only once except when a P/ occurs in the constant. A new word is added to the pool for every constant containing a P/. (P/ is defined in paragraph 4.2.8.)

4.2 TYPES OF POOL CONSTANTS

4.2.1 Decimal (Fixed Point)

D/±d''' d.d''' dE±p''' p Txx

where:

d's are decimal digits.

p''' p is the decimal power of 10 by which the number is multiplied. (If E ± p''' p is omitted, zero is assumed.)

xx is a 1- or 2- digit decimal number which denotes the bit position (0 through 39) after which the binary point is assumed to follow. (If Txx is omitted, 39 is assumed.)

Example:

D/ -63.8E4

If a sign is omitted, plus is assumed. If the decimal point is omitted, an integral value is assumed. It is not necessary for a digit to appear to the left of the decimal point.

4.2.2 Floating Point Decimal

D/±d''' d.d''' d F± p''' p

where d's and p's are the same as for decimal. The F may not be omitted since this distinguishes floating point from fixed point. If no digit follows F, zero is assumed. Normalized numbers are produced.

Example:

D/397.46F

4.2.3 Word

W/c''' c

Where c's are SOLOMON characters. There are exactly 6 c's in this constant.

Example:

W/CØST ␣␣

4.2.4 Alphanumeric

A/ c''' c

where c's are SOLOMON characters other than plus, comma, semicolon, and dollar sign. If fewer than 6 characters are written before a plus, comma, semicolon, or dollar sign, the constant word is left justified and then right filled with zeros.

Example:

A/MØUNT

4.2.5 Octal

Ø/o''' o Txx

where o's are octal digits. If fewer than 13 digits are written, the word is left filled with zeros. The bit position (0 through 39) which contains the last binary digit of the converted value is specified by xx (if Txx is omitted, a value of 39 is assumed).

Example:

Ø /374T20

4.2.6 Hexadecimal

H/ h''' h Txx

where h's are hexadecimal digits. The digits for the binary configurations 1010 through 1111 are J, K, L, M, N, and Ø. If fewer than 10 digits are written, the word is left filled with zeros. The bit position (0 through 39) which contains the last binary digit of the converted value is specified by xx (if Txx is omitted, a value of 39 is assumed).

Example:

H/76L4K2T18

4.2.7 Binary

B/b''' b. nn Txx

where b's are binary digits and nn is the number of times the configuration b''' b occurs. If .nn is omitted, 1 is assumed. The bit position (0 through 39) which contains the last digit is specified by xx (if Txx is omitted, 39 is assumed).

Example:

B/101.5T22

4.2.8 Parameter

P/ symbol, Txx

where symbol is any permissible base adr form other than a constant.

These are:

a.	Pure decimal	(e.g., - 7683)
b.	Octal location	(e.g., - M/76334)
c.	Symbol	(e.g., - ALPHA)
d.	Combination	(e.g., - ALPHA + M/76335-7683)

P/ is converted to its binary equivalent (e.g., the definition of the symbol, in case 3). The bit position (0 through 39) which contains the last digit is specified by xx. The comma and the Txx may not be omitted.

Example:

P/ALPHA, T30

4.3 COMPOSITE

Any of the constants except Decimal, Floating Point Decimal, and Word may be made composite (i.e., several constants of the same or different type may be converted into a single machine word). The Decimal constant may be included in a composite constant if it is integral and positive.

The various parts of the constant are separated by plus signs.

Example:

Ø/73T12+P/ALPHA + BETS, T30 + D/10

Note that a plus sign is permitted in a parameter constant, but, since it occurs between the slash and comma, it cannot be confused with the packed constant indicator.

5. PSEUDO-INSTRUCTIONS

Any legitimate entry in the INST Field of the SOLOMON assembly coding form belongs to one of two classes: machine instruction (executable during run time) and nonmachine instruction. A nonmachine instruction is called a pseudo-instruction. These instructions are not executable by the assembled program. Pseudo-instructions do not cause machine instructions to be generated although machine constant words may be generated.

5.1 GENERAL

In what follows anything appearing in a box preceding the pseudo-instruction will appear in the location field, anything appearing in a box following the pseudo-instruction will appear in the address field. All commas in pseudo-instruction formats are critical, and arg can be any legitimate form of an address as defined in Section 3.

5.2 DETAILED INSTRUCTION DESCRIPTIONS

Location	Instruction	Address
LOC	SEG	START, END

The SEG (segment) pseudo-instruction is used to define the block of memory and a tape record into which the program is to fit. START defines the memory location of the first line of the program, and END defines the location of the last line. The value of the Location Counter will not exceed the value of END, therefore, effectively reserving a block of memory size - END locations.

If START and/or END are left blank, then the following standard values are assumed by the assembler:

If START = ␣ , it is assumed that START = the lowest available memory location after allocation has been made for the loading routine and any constants needed by the program.

If END = ⊔ , it is assumed that END = the location of the machine instruction preceding the next SEG or END pseudo-instruction.

The simplest form of SEG is;

Location	Instruction	Address
	SEG	

Here START and END = ⊔ , so the standard assumptions will be made and END - START words is the segment length in memory and the record length on tape.

There are three other ways SEG could appear, they are:

Location	Instruction	Address
		START ≠ ⊔
	SEG	START (i.e., END = ⊔)
	SEG	START, END START ≠ ⊔ (i.e., END ≠ ⊔)
	SEG	, END START = ⊔ (i.e., END ≠ ⊔)

LOC is normally blank but if an argument appears in the LOCATION Field, this defines LOC = START

If no SEG card is used by the programmer, then the assembler will assume an LSEG card (See LSEG)

Location	Instruction	Address
LOC	LSEG	START, END, TAPE #, NAME, WHERE

LSEG (Load Segment) is the same as SEG except that it results in an automatic loading routine which will load the tape records into memory as opposed to SEG where the programmer is responsible for producing the loading routine. The three additional sub fields in the ADDRESS Field serve the following purposes:

TAPE # is the programmer's tape number. If TAPE # = ⊔, then some standard assumption is made (e.g., Tape #1).

NAME is the identification label for each tape record so that by searching the tape, the appropriate segment may be selected and read into memory. If NAME = ␣, it is assumed to be 12 spaces.

WHERE is the location to which the machine transfers for its next instruction after the tape record is loaded. If WHERE = ␣, it is assumed that WHERE = START.

The following table summarizes the standard assumptions made if arguments are blank.

IF	THEN IT IS ASSUMED THAT
TAPE # = ␣	TAPE # = some established number such as TAPE #1
NAME = ␣	NAME = twelve blank spaces
WHERE = ␣	WHERE = START

EXAMPLES of possible LSEG pseudo-instructions

Location	Instruction	Address
	LSEG	

All standard assumptions are made (i.e., ADDRESS Field contains 5 blanks

LESG 792, ALPHA, , , 1001

The segment will start at location 792 and end at location ALPHA (which may have been previously defined or may be defined before the next SEG or LSEG). The standard tape number will be assumed, the tape records name will be ␣ ␣ ␣ ␣ ␣ ␣ ␣ ␣ ␣ ␣ ␣ ␣ , and a transfer is made to location 1001 after the tape is automatically loaded.

Location	Instruction	Address
	LSEG	, , , , 2000

All standard assumptions are made except that a transfer to location 2000 will take place after the tape is loaded into memory.

Location	Instruction	Address
	NAME	ARG

In writing a program which requires many symbols, the programmer's ingenuity may be taxed by having to invent mnemonics which are both unique and suggestive of their function in the program.

The pseudo-instruction NAME acts as a barrier in the program on either side of which the same symbol may be used for a different address. Each barrier defines a Name Block whose name is the symbolic argument ARG (eight characters or less) which precedes it.

Suppose, for example, a programmer has written a program with two Name Blocks (two NAME pseudo-instructions) with names JOE and SAM and he has used the symbolic argument DIST in each Name Block. If he is in block SAM, he can refer to DIST as defined in block SAM by writing DIST. If he wishes to refer to DIST as defined in block JOE, he must write JOE. DIST (i.e., DIST as defined in Name Block JOE).

His coding would be similar to the following:

Location	Instruction	Address
	NAME	JOE
BETA	ADD	ALPHA,S; DIST
	NAME	SAM
DELTA	MUL	DIST; GAMMA,S
ZETA	ADD	DIST; JOE DIST, S

NAME establishes a barrier between all symbolic addresses but not program point and regional addresses. Furthermore, there is no intrinsic relationship between NAME and SEG or LSEG.

Location	Instruction	Address
	REG	X, ARG, #

This pseudo instruction is used to define a region. This is done by assigning the base (first line) of the region to a defined memory location. The base of region X is assigned to location ARG, the regions's 0 position. # is the number of lines in the region, if # = ␣, (i.e., contains a blank, the region

is unbounded). The region is referred to modulo #, except where # = ␣ , when the region is referred to modulo the memory size.

It must be kept in mind that a REG pseudo-instruction does not reserve space in memory. If the program counter encounters the region while stepping through memory locations, it will continue to assign lines to the locations in the region. A regional address is defined if and only if the region in question is defined by a REG pseudo-instruction.

Location	Instruction	Address
	END	

End of assembly. If no END card is used, the assembler will generate one and an error indication will be given.

Location	Instruction	Address
LOC	BSS	#

BSS (block started by symbol) is a way of reserving a number of consecutive lines in a segment which will not have instructions placed in them (as contrasted with REG).

If the programmer, for example, needs to reserve the next 50 words, he may write

Location	Instruction	Address
	BSS	50

When encountered, this instruction will cause the location counter to be increased by $(50)_{10}$

If it is desired to give the name ALPHA to the first word of the block, the instruction can be written

Location	Instruction	Address
ALPHA	BSS	50

With BSS, it is not possible to associate location symbols with any words of the reserved block except the first word.

In general it can be said that when BSS is encountered during the assembly, the LOCATION Counter is incremented by #, effectively skipping # consecutive memory locations. LOC is the name assigned to the first line of the block.

Location	Instruction	Address
LOC	BES	#

A BES (block ending with symbol) pseudo-instruction is another way of reserving a block of memory. The difference between BSS and BES is that where LOC refers to the first line of the reserved block in a BSS; LOC refers to the last line of the reserved block in a BES.

Location	Instruction	Address
LOC	DEC	#'s

DEC (decimal) causes the decimal numbers specified by # in the address field to be converted to binary and assigned to successive locations beginning with the current value of the Location Counter. If there is a symbol, LOC, it is entered in the dictionary with the current value of the Location Counter. The first, (left most decimal number specified in the ADDRESS Field) can be referred to by this location symbol.

EXAMPLE:

Suppose the value in the Location Counter is 3900 when the following instruction is encountered:

Location	Instruction	Address
ALPHA	DEC	1,-3,5,7,9

The effect of this instruction is to enter the symbol ALPHA in the dictionary with the value 3900. The five integers 1, -3, 5, 7, 9 are converted to binary and assigned to locations 3900, 3901, 3902, 3903, and 3904, respectively. The value of the Location Counter upon completion will be $(3905)_{10}$.

If the programmer wishes to use the decimal number 9 from the above list as an argument in another instruction he can write

Location	Instruction	Address
	ADD	ALPHA + 4; BETA

The decimal numbers are separated by commas and are converted to appropriate internal form (floating point or fixed point binary) and stored in consecutive memory locations beginning with location LOC. The form in which decimal numbers may be written is described under pool constants except that the D/ is omitted.

Location	Instruction	Address
LOC	OCT	#'s

OCT is essentially the same as DEC in that a list of numbers is converted to binary and stored in consecutive memory locations starting with LOC. However, there are four differences between DEC and OCT. In OCT;

a. Octal numbers are used

b. All numbers are fixed point

c. No exponentiation is used

d. Fractions and negative numbers are not allowed.

(Note: See Pool Constants, Ø/)

EXAMPLE 1

Location	Instruction	Address
ALPHA	OCT	55T30

The octal number 55 would be stored in location ALPHA and terminate at bit position 30.

Bit Position 0 1 2 - - - - 25 26 27 28 29 30 - - - - 39

Value 0 0 - - - - 1 0 1 1 0 1 - - - -

EXAMPLE 2

Location	Instruction	Address
GAMMA	OCT	33T16, 472, 37T11

This list would be stored as follows:

Location	Contents	Final Bit Position
GAMMA	011011	16
GAMMA + 1	100111010	39
GAMMA + 2	011111	11

Location	Instruction	Address
LOC	HEX	#'s

HEX is essentially the same as OCT, except that hexadecimal numbers are used (i.e., 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, J, K, L, M, N, Ø).

(Note: See Pool Constant, H/)

EXAMPLE 1

Location	Instruction	Address
BETA	HEX	ØT31

The hexadecimal number would be stored in location BETA as follows:

Bit Position 0 1 - - - - 28 29 30 31 - - - 39

Value 0 0 - - - - 1 1 1 1 - - - 0

EXAMPLE 2

Location	Instruction	Address
DELTA	HEX	3JT21, 1KT20, 44T33

The list would be stored as follows:

Location	Contents	Final Bit Position
DELTA	00111010	21
DELTA + 1	00011011	10
DELTA + 2	01000100	33

Location	Instruction	Address
LOC	ALPH	CHARACTERS $

Following the pseudo-instruction ALPH (alphanumeric) will be a sequence of characters (including spaces) followed by a dollar sign ($). Each of these characters except the $ will be changed to its standard 6-bit configuration (output form) and each group of 6 characters stored in a 40-bit memory

location starting with bit No. 0 (i.e., left justified). The dollar sign, $, marks the last character in the sequence and the rest of the line following $ may be used as usual, for comments. If the number of characters is not a multiple of 6, the remainder of the list word will be filled on the right with spaces. Continuation cards may be used.

Location	Instruction	Address
LOC	ALPH 7	CHARACTERS

This instruction is identical to ALPH except that characters are changed into the standard seven-bit configuration. This, of course, limits the programmer to 5 characters per word with 5 bits left over, but continuation cards may be used.

Location	Instruction	Address
LOC	TABLE	#'s

TABLE is a way of composing a table of arbitrary terms. The first word of the table is named by whatever appears in LOC.

The body of the table is made by converting the words, separated by commas, from the ADDRESS Field into consecutive machine words. The words in the ADDRESS Field are written in exactly the same form as pool constants (q.v.) including the conversion indicator (B/, D., etc). Packed con stants are permitted as is P/. The constants appearing in the ADDRESS Field of a TABLE pseudo-instruction are not pooled.

Continuation cards may be used.

Location	Instruction	Address
LOC	LIB	NAME

This instruction (Library Call), though listed among the pseudo-instructions, is actually a hybrid in that it is converted to a running program instruction. At the end of an assembly segment, this pseudo-instruction (which may appear anywhere in the segment) calls subroutine NAME from the library. The pseudo-instruction is then converted to a subroutine call, which is stored in location LOC, in the running program; it is the programmer's responsibility to fix the data, linkage, etc, for the subroutine call.

EXAMPLE:

Location	Instruction	Address
SIGMA	LIB	COSINE

Location	Instruction	Address
	LIST	
	.	
	.	
	.	
	.	
	.	
	UNLIST	

These two pseudo-instructions LIST and UNLIST will be discussed together in that they are analogous to left and right hand parentheses around a block of coding.

All lines included between LIST and UNLIST, respectively, will appear in the listing; all lines included between UNLIST and LIST, respectively, will not appear in the listing. If no LIST or UNLIST cards are used by the programmer, then the assembler will assume a LIST card before the first word of the program and an UNLIST card after END. In other words, the normal mode of assembly is to produce a listing.

Included in a normal listing will be a print out of the symbol table following the listing of lines of code. If END is within the scope of LIST (included within the parentheses), the symbol table will be printed out. This, of course, includes the case where no LIST or UNLIST cards are used.

If END is within the scope of UNLIST, the symbol table will not be printed out.

EXAMPLE 1

Location	Instruction	Address
	.	
	.	
	.	
ALPHA	ADD	BETA;GAMMA,S
	UNLIST	
	.	
	.	
	.	
JOE	SUB	KAY;MARY, S
PETE	ADD	GAMMA;MARY, S
	LIST	
	.	
	.	
	.	
	END	

Lines JOE and PETE will not appear in the listing and the symbol table will be printed out.

EXAMPLE 2

Location	Instruction	Address
FIRST LINE OF PROGRAM---------->		
	.	
	.	
	.	
	.	
	END	

All lines and the symbol table will appear in the listing.

EXAMPLE 3

Location	Instruction	Address
First line of program--------	UNLIST	
	.	
	.	
	.	
	LIST	
SAM	MUL	ZETA;IOTA
SUM	SUB	KAPPA;SAM
	UNLIST	
	.	
	.	
	.	
	END	

Lines SAM and SUM will be the only lines in the listing and the symbol table will not be printed out.

Location	Instruction	Address
	SRLIST	

In the normal assembly mode, library subroutines do not appear in the listing. If the programmer desires to have subroutines listed, he can achiev this by using the SRLIST (subroutine list) pseudo-instruction.

EXAMPLE:

Location	Instruction	Address
	SRLIST	
ALPHA	LIB	COSINE

The subroutine for COSINE will appear in the listing.

SRLIST may appear anywhere in the program and will result in the listing of all subroutines.

Location	Instruction	Address
	DEF	

By using a DEF (define symbol) pseudo-instruction, the programmer instructs the assembler to assign all undefined symbols in the program to locations in memory immediately following the program.

It must be kept in mind by the programmer that if DEF is used, no error indication will be given for the appearance of an undefined symbol in the ADDRESS Field, since indeed there are no undefined symbols.

Location	Instruction	Address
LOC	EQU	ARG

The undefined symbol LOC is defined as equivalent to ARG. ARG may be an absolute, regional, or symbolic address. If the programmer at any point in the program realizes that memory locations may be conserved by assigning two or more symbols to the same location at different points in the program, he may use EQU.

Another example would be its use as a name for something, the precise nature of which the programmer does not yet know. For example, he might wish to refer to some instruction which he has not written down and does not yet want to decide on the name of this instruction. He may be at a point such that based on the result of a test, he wishes to assign the line to the symbolic address GOOD or NOGOOD. The programmer can put (for example) TEST in the LOCATION Field and later, when the result is known to him, write:

Location	Instruction	Address
NOGOOD	EQU	TEST

Location	Instruction	Address
	ASSYLB	NAME

If the programmer wishes to insert a program into the library, he must use this pseudo-instruction as the first line of the program. The program will be assembled into library format, added to the library, and given the subroutine name NAME.

EXAMPLE:

A programmer may write a subroutine to extract a fifth root, and wants to put it on the library tape. Assume he wants to name this subroutine FIFRT; he would begin his program with:

Location	Instruction	Address
	ASSYLB	FIFRT

In all later assemblies (using this library), he may call this routine by writing:

	LIB	FIFRT
Location	Instruction	Address

EXEC

The assembled program will operate under an executive system. EXEC must appear in the first line of the program.

Location	Instruction	Address
	LANDGO	

A LANDGO (load and go) pseudo-instruction will cause the execution of the running program immediately following assembly, provided no serious errors are detected.

Only one of the pseudo-instructions ASSYLB, EXEC, or LANDGO may appear in a program, and if one does appear, it must be the first card of

such a program. Certain other pseudo-instructions which must be "at the beginning of a program" follow one of the above three, but must precede any nonpseudo-instruction.

Location	Instruction	Address
	LON	

This pseudo-instruction will result in the production of a listing only, (no machine code output from the assembly). LON (listing only) ignores UNLIST and will list the names of subroutines called but will not actually call them. LON is used by the programmer for debugging.

If during the assembly an error indication is given which results in a line being treated as a comment (see Section 2), then the remainder of the assembly will be as if a LON pseudo-instruction were inserted after the error indication. This feature is desirable because an error resulting in a line being treated as a comment generally generates additional errors in the program, which makes output unusable.

Even though LON produces an exact listing of lines of code, the line containing LON itself will be slightly altered in the listing. The line containing LON will appear in the listing with an X in its T field. This is necessary because if the listing, at some later time, is used as input, the line containing LON must be ignored by the assembler or no output will result. Since an X in the T field means, "ignore this line," it is automatically affixed to a LON pseudo-instruction after the first time the assembler encounters it.

The programmer writes:

T	Location	Instruction	Address
		LON	

but sees in the listing:

T	Location	Instruction	Address
X		LON	

Location	Instruction	Address
	TRACE	

The effect of TRACE is that each instruction in the running program, its location, the contents of all affected registers, etc, will be put out on tape. Tracing is used as a debugging technique. Inasmuch as subroutines are assumed to be debugged, TRACE will not trace a subroutine.

Just as in the case of LON, an X is inserted by the assembler in the T field of a line containing TRACE. That is:

T	Location	Instruction	Address
		TRACE	

will appear in the listing as:

T	Location	Instruction	Address
X		TRACE	

Location	Instruction	Address
	UNTRACE	

This is the counter instruction for TRACE, (if the programmer wishes to trace only a block of lines, bracketing these lines between a TRACE and an UNTRACE results in the desired operation).

Just as in the case of LON, an X is inserted by the assembler in the T field of a line containing UNTRACE.

EXAMPLE:

T	Location	Instruction	Address
		TRACE	
	ALPHA	ADD	JILL;JACK, S
	BETA	MUL	DELTA;SAM, S
		UNTRACE	

Lines ALPHA and BETA will be traced and the listing of these lines will be:

T	Location	Instruction	Address
X		TRACE	
	ALPHA	ADD	JILL;JACK, S
	BETA	MUL	DELTA;SAM, S
X		UNTRACE	

Location	Instruction	Address
	FILL	CHARACTERS

A forward sequence break (see paragraph 1.2.2) causes the program counter to skip over a block of memory locations. The programmer may fill these locations as he wishes with a FILL pseudo-instruction.

FILL is followed by any one of, but not a combination of, these 3 pool constants; Ø/, B/, or H/ (see Section 4) which are assembled into a 40-bit binary word and stored in a special memory location in the assembler which could be called word FILL. Now, every time a forward program break instruction is encountered, all of the skipped locations will be filled with the current value of FILL.

EXAMPLE:

Assume the programmer wishes to fill all skipped memory locations with ones; in the beginning of his program he would write

Location	Instruction	Address
	FILL	B/1.40

If a BSS pseudo-instruction is wanted to increment the program counter by 10 and the programmer wishes to fill the first 5 skipped locations with zeroes and the last five with 1's, he must use 2 BSS instructions and change the current value of FILL. His coding would appear as follows:

Location	Instruction	Address
	FILL	B/0.40
ALPHA	BSS	5
	FILL	B/1.40
BETA	BSS	5

Location	Instruction	Address
	PATCH	Tape Unit, START, END

PATCH (Combine Patches of tape records) enables the programmer to use as assembler input parts of previously taped programs.

Assume, for example, a programmer wishes to assemble a 300-line program, the first 100 lines of which are new coding, the next 100 lines are identical to lines 233 through 332 of a previously assembled program, the next 50 lines 128 through 177 of another previously assembled program. Assume that the 2 previous tapes are mounted on tape units 1 and 2; then the card reader would read the following code:

Location	Instruction	Address
FIRST	.	.
FIRST	.	.
.		
.		
.		
FIRST + 99	.	.
	PATCH	1,233,332
NEXT	.	.
NEXT + 1	.	.
.		
.		
.		
.		
NEXT + 49		
	PATCH	2,128,177
	END	

Blocks of lines can be patched in any order (a block of lines 326 through 409 can be called before a block from the same tape, lines 25 through 48, and there is no limit to the number of times patches may be read from a tape)

If a single line is to be patched then START = END = desired line in the pseudo-instruction.

When the cards with collated patches are all written onto the listing tape, PATCH pseudo-instructions will appear with X in the Type Field.

Location	Instruction	Address
	CARDS	n

If a CARDS pseudo-instruction card is inserted at the beginning of the card deck, the program will assemble 2 to 3 times faster.

CARDS pseudo-instructions will appear in the listing with an X in the Type Field.

When a CARDS pseudo-instruction is used, the assembly system uses two or more tapes for output from the read-in phase of assembly. Each tape contains the information from an approximately equal number of cards. Rewinding of the first tapes written can then proceed concurrent with writing of later tapes.

5.3 SUMMARY OF PSEUDO-INSTRUCTION CODES

SEG	Segment
LSEG	Load Segment
NAME	Name
REG	Region
END	End
BSS	Block Started by Symbol
BES	Block Ended by Symbol
DEC	Decimal
OCT	Octal
HEX	Hexadecimal
ALPH	Alphanumeric, 6 bits
ALPH 7	Alphanumeric 6 bits per character
TABLE	Table
FILL	Word Fill
EXEC	Executive System

UNTRACE	Will not Trace
TRACE	Trace
LANDGO	Load and Go
LON	List Only
ASSYLB	Assemble into Library
EQU	Equivalent
DEF	Define
SRLIST	Subroutine Listing
LIST	Make Listing
UNLIST	Make no Listing
LIB	Library Call
PATCH	Patch
CARDS	Cards