SPERRY RAND

UNIVAC

1106
SYSTEM

1108
MULTI-PROCESSOR
SYSTEM

ASSEMBLER

PROGRAMMERS
REFERENCE

UP-4040 Rev. 2

This manual is published by the Univac Division of Sperry Rand Corporation in loose leaf format. This format provides a rapid and complete means of keeping recipients apprised of UNIVAC® Systems developments. The information presented herein may not reflect the current status of the programming effort. For the current status of the programming, contact your local Univac Representative.

The Univac Division will issue updating packages, utilizing primarily a page-for-page or unit replacement technique. Such issuance will provide notification of software changes and refinements. The Univac Division reserves the right to make such additions, corrections, and/or deletions as, in the judgment of the Univac Division, are required by the development of its Systems.

UNIVAC is a registered trademark of Sperry Rand Corporation.

PRINTED IN U.S.A

CONTENTS

TABLES

1. BASIC ASSEMBLER LANGUAGE

1.1. INTRODUCTION

The UNIVAC 1106/1108 Assembler System is a symbolic coding language allowing simple, brief expressions as well as complex expressions. The assembler provides rapid translation from this symbolic language to machine-language relocatable object coding for the UNIVAC 1106 System and the UNIVAC 1108 Multi-Processor System.

The assembler operates under control of the EXEC 8 Operating System. The outputs of the assembler are made consistent with the system by using standard interfacing routines both for the source files and the relocatable program generated. (See Appendix H for differences between the processor herein described and the assembly processor operating under the EXEC II Operating System.)

The assembly language includes a wide and sophisticated variety of operators which allow the fabrication of desired fields based on information provided at assembly time. The instruction function codes are assigned mnemonics which describe the hardware function of each instruction. Assembler directive commands provide the programmer with the ability to generate data words and values based on specific conditions at assembly time. Multiple location counters provide a means of preparing for program segmentation and controlling address generation during assembly of a source code program.

The assembler produces a relocatable binary output for processing by the loading mechanism of the system. If requested, it supplies a side-by-side listing of the original symbolic coding and/or an edited octal representation of each word generated. Flags indicate error in the symbolic coding detected by the assembler.

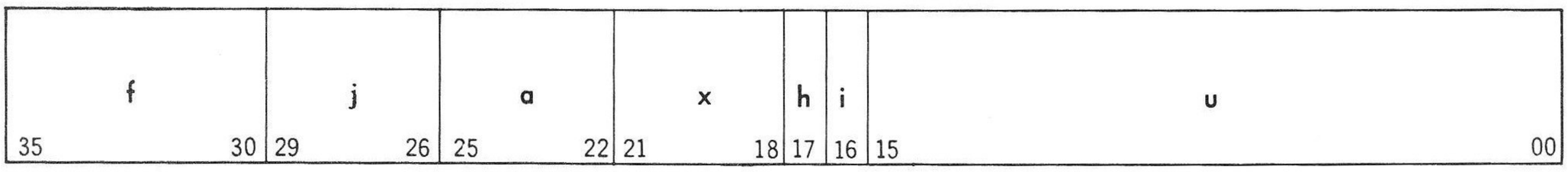
1.2. SYMBOLIC CODING FORMAT

In writing instructions using the assembler language, the programmer is primarily concerned with three fields: (1) a label field, (2) an operation field, and (3) an operand field. It is possible to relate the symbolic coding to its associated flowchart, if desired, by appending comments to each instruction line or program segment.

All of the fields and subfields following the label field in the assembler are in free form providing the greatest convenience possible for the programmer. Consequently, the programmer is not hampered by the necessity to consider fixed form boundaries in the design of symbolic coding.

1.2.1. Computer Instruction Word Format

The assembly program recognizes the set of mnemonic instructions representing the machine code instructions listed in Appendix A of this manual. The format of these machine code instructions is as follows:



f indicates the function code.
j indicates the partial word designator or minor function code.
a indicates the control register or input/output channel.
x indicates the index register.
h indicates index modification.
i indicates indirect addressing.
u indicates the address field.

1.2.2. Assembler Format

A basic line of coding consists of a label field, operation field, and operand field. Each field is terminated by one or more spaces, and may be divided into subfields with a comma terminating each subfield. The last subfield in a field is terminated by at least one space.

Two optional formats of a symbolic instruction are presented below.

	LABEL FIELD	OPERATION FIELD	OPERAND FIELD
Option I		F,J	A,U,X
Option II		F	A,U,X,J

The entry in the operation field is the instruction mnemonic.

The entry in the A subfield represents the absolute address of an arithmetic, index, or R register as required by the instruction.

The entry in the U subfield represents the operand base address. Indirect addressing is indicated by means of an asterisk preceding the U subfield, for example, *U.

The entry in the X subfield represents the specific index register to be used. Index register modification is indicated by means of an asterisk preceding the X subfield, for example, *X.

The J subfield is used only to designate partial word transfers to and from the U subfield. It has two optional positions: option I permits the J subfield to follow the instruction mnemonic; and J designator follows the X subfield in option II. Option II may not be used with line items (see 1.5.5).

Example:

```
      1
      LABEL  Δ  OPERATION  Δ  OPERAND
1.    LGP       LA         12,  TAG,,4.
2.    PRSS      LA,6       14,  TAG2.
3.              LA         20,  TAG3,10
```

Explanation:

Line 1:

Line 1 uses option II format. LGP is the label starting in column 1. The operation field contains the instruction mnemonic LA indicating the use of an arithmetic register. The operand field uses the absolute arithmetic register address 12; the U subfield is TAG. Indexing is not required so the construction comma space comma (,␣,) is used preceding the partial word designator 4.

Line 2:

Line 2 uses option I format. The partial word designator 6 follows the instruction mnemonic. The absolute address of arithmetic register 14 is in the operand field.

Line 3:

Since a label is not used, the instruction mnemonic may start in column 2. The operand field contains the absolute address of arithmetic register 20. Indexing is indicated from index register 10.

1.2.3. Mnemonic Designators

Registers may be addressed by using either the absolute register addresses or by using mnemonic designators. When using instructions which require an index register, the programmer uses absolute addresses 0 through 11 in the A subfield. If the operation field indicates an arithmetic register, absolute addresses 12 through 27 are used. The special registers (R registers) require 65 through 79 in the A subfield.

At systems generation time, most 1106/1108 systems are equipped with the systems procedure AXR$. This procedure enables the programmer to use the mnemonic designators of the registers and partial word designators.

When using an instruction which requires an arithmetic register, the mnemonic designators A0 through A15 may be used in the A subfield of the coding line. The special registers (R registers) use mnemonic designators R1 through R15.

When the instruction mnemonic specifies the use of an index register, the mnemonic designators used are X0 through X11. Do not use register X0 as the contents of that register are destroyed whenever an executive interrupt occurs (see Appendix C.1). Table 1-1 cross-references the mnemonic designators with the absolute addresses of all control (A,X,R) registers.

INDEX REGISTERS		ARITHMETIC REGISTERS		R REGISTERS		PARTIAL WORD DESIGNATION	
MNEMONIC	ABSOLUTE	MNEMONIC	ABSOLUTE	MNEMONIC	ABSOLUTE	MNEMONIC	ABSOLUTE
X0	0	A0	12	R1	65	W	0
X1	1	A1	13	R2	66	H2	1
X2	2	A2	14	R3	67	H1	2
X3	3	A3	15	R4	68	XH2	3
X4	4	A4	16	R5	69	XH1	4
X5	5	A5	17	R6	70	T3	5
X6	6	A6	18	R7	71	T2	6
X7	7	A7	19	R8	72	T1	7
X8	8	A8	20	R9	73	S1	15
X9	9	A9	21	R10	74	S2	14
X10	10	A10	22	R11	75	S3	13
X11	11	A11	23	R12	76	S4	12
		A12	24	R13	77	S5	11
		A13	25	R14	78	S6	10
		A14	26	R15	79	Q1	7
		A15	27			Q2	4
						Q3	6
						Q4	5
						U	16
						XU	17

Table 1-1. Mnemonic Designation and Absolute Addresses of Control Registers

Example:

```
1          LABEL     Δ     OPERATION     Δ     OPERAND
1.     LA        A0,       TAG.
2.     LA        12,       TAG.
3.        LXM       X5,          15.
4.        LXM       5,           15.
```

Explanation:

Line 1 is the same as Line 2.

Line 3 is the same as Line 4.

The above description reflects the normal machine code format. If, however, the J designator is octal 016 or 017 and the following conditions exist, then the U field is considered to be 16 bits in length (bit positions 17–00). Instruction addresses under these conditions will be taken as 16-bit quantities and the binary instruction word generated accordingly.

J = 016 or 017	(immediate address)
X = 0	(no index)
h = 0	(no index incrementation)
i = 0	(no indirect addressing)
f < 070	(J is not minor function code)

1.2.3.1. Partial Word Mnemonics

When the AXR$ procedure is initiated, it is possible to use mnemonic designators when indicating partial word transfers.

Example:

```
      1    LABEL    Δ    OPERATION    Δ    OPERAND
1.         LA          A15,          REN,  S3.
2.         LA          27,           OIR,  13.
3.            AA       A2,T3         PIC.
4.            AA       14,5          PIC.
```

Explanation:

Lines 1 and 2 are equivalent.

Lines 3 and 4 are equivalent.

An additional option is provided in that four special instructions involving arithmetic, index, or R registers may be coded without indicating the appropriate A, X, or R. In this case, the value of the A field determines which register is to be used, as shown in the following example. The assembler inserts the appropriate instruction into the coding.

Example:

```
      1 LABEL     Δ     OPERATION     Δ     OPERAND
1.      L           A5,           RTB43
2.      L           X2,           WSRIME
3.      L           R7,           TELE
4.      S           A9,           SP9
5.      A           X4,           QXZ
```

Explanation:

Line 1 is equivalent to LA A5, RTB43.

Line 2 is equivalent to LX X2, WSRIME.

Line 3 is equivalent to LR R7, TELE.

Line 4 is equivalent to SA A9, SP9.

Line 5 is equivalent to AX X4, QXZ.

1.3. DESCRIPTION OF FIELDS

The programmer is primarily concerned with the label field, operation field, and operand field. The label field must start in column 1. The fields following the label field are freeform and may start in column 2 if there is no label field.

1.3.1. Label Field

The label field is optional. When used, the label field must start in column 1. No other field may start in column 1. The label field may contain a declaration of a specific location counter, a label, or both. The label field is terminated by a blank.

1.3.2. Labels

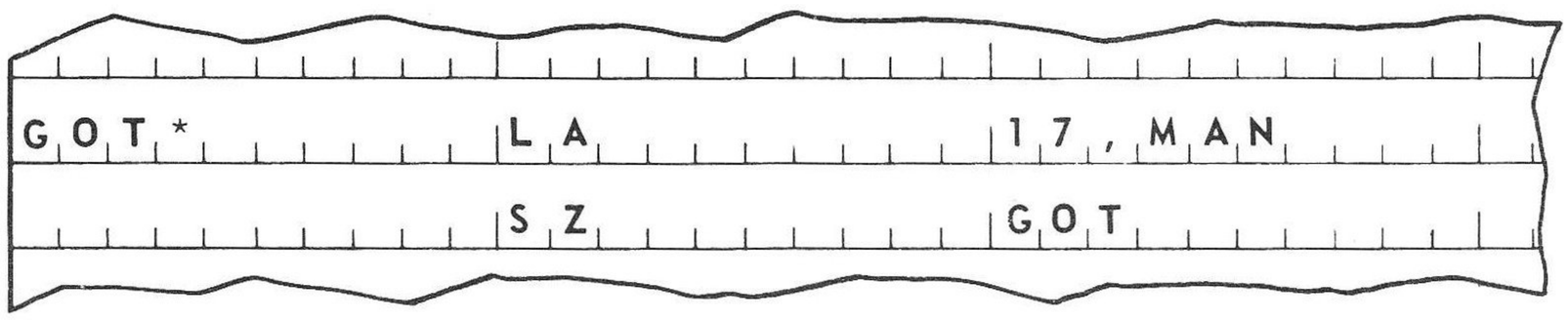
A label identifies a value or a line of symbolic coding. When a label is used, the assembler assigns it a relative address which is the value of the current controlling location counter. A relative address is not assigned to a label used with assembler directives EQU, NAME, FORM, PROC, DO, INFO, NEG, LIT, (see Section 2).

A label consists of one to six alphanumeric characters starting with an alphabetic character in column 1.

1.3.3. Externalized Labels

An external label is a label the value of which is known outside the program. Such labels are suffixed with an asterisk as is GOT* in the following example. The asterisk does not count as a character of the label. Any label which is assigned a single precision value including locations of double precision constants may be externalized. They are assigned the relative address of the first word of the value generated.

Example:



1.3.4. Subscripted Labels

The assembler permits usage of subscripted labels. A subscript may be any legitimate assembler item, an expression, or another subscripted label. A label may not be used as its own subscript.

A subscript is enclosed in parentheses immediately following the label with no intervening spaces between the label and subscript. If more than one subscript is used, each subscript is separated by a comma. A series of subscripts is enclosed in parentheses.

Examples:

```
L(3)
L(4,1)
L(1,M(1))
L(I(1),J(1),K(1))
L(2,P(1,R(1,1)+017))
L(4,3,SIZE//2)
```

The asterisk of an externally defined subscripted label precedes the left parenthesis of the subscript, for example, DOG*(1). The asterisk, parenthesis, and subscripts are not counted as characters of the label. Subscripted labels may be redefined in a program without producing an error flag.

1.3.5. Operation Field

The operation field starts with the first nonblank character following the label field and is terminated by a blank except as noted below. If no label is used, the operation field can start in column 2, the blank in column 1, indicating the end of the label field. The contents of the operation field may be any one of the following:

- An instruction mnemonic, with a possible J designator.
- A + or a - sign indicating a data word of octal, decimal, or alpha designation. In this case a space is not necessary to terminate the operation field. That is, the operand or the value may follow the + or - immediately. For example, +2 is equivalent to +b2.
- An assembler directive.
- A label previously defined as a legitimate entry point to a PROC or the label on a FORM directive line. (See Section 3.)
- An alphanumeric constant enclosed in apostrophes without a leading sign.

If the operation field contains an assembler directive other than RES (which changes the location counter) or DO (which may generate object code), the location counter is not affected. In all other cases, the controlling location counter is incremented by the number of words generated after the line has been processed.

Example:

```
    1 LABEL     Δ OPERATION     Δ OPERAND
1.    LXI         X5,5
2.    +8
3.        `ROSS´
4.    END
```

Explanation:

Line 1:
The operation field contains the instruction mnemonic LXI.
Line 2:
The operation field contains the data word 8.
Line 3:
The operation field contains the alphabetic item 'ROSS'.
Line 4:
The operation field contains the assembler directive END.

1.3.6. Operand Field

The operand field starts with the first nonblank character following the operation field. The components of the operand field are called subfields and represent the information necessary to complete the type of line determined by the operation field. Subfields are separated by commas. A comma may be followed by one or more blanks.

It is not necessary for the operand field to contain the maximum number of subfields implied by the operation field. When omitting a subfield, other than the normal first or last subfield, the construction, comma zero comma (,0,) or two contiguous commas (,,) is necessary.

If the last subfield is omitted, a comma is not required to appear after the last coded subfield. A space period space (b.b) coded after the last subfield stops the assembler scan and reduces assembly time.

Example:

```
      1 LABEL     Δ  OPERATION     Δ  OPERAND
1.    TETH        LA  A5,  TAG,,S6.
2.                LX  X2,  *TAG.
3.    BLR         L   A9,  FIL,,Q3.
4.    END.
```

1.3.7. Location Counter Declaration

There are 0-31 location counters in the assembler. Any location counter may be used or referenced in any sequence. These counters provide information required by the collector to regroup lines of coding in any specified manner. The regrouping capability enables isolation of instruction components giving flexibility in segmentation.

A program remains under control of location counter zero if no location counter is explicitly specified. When a specific location counter is specified, all subsequent coding is under its control until another location counter is explicitly specified.

$(e) written as the first entry in the label field declares a specific location counter. The e is any location counter 0 through 31.

A location counter designation may precede a label. The label must immediately follow the location counter designation. The format is $(e), LABEL with no intervening blanks.

Example:

```
1.                  LA      A4,18,S2
2.                  J       TAG
3.    $(2),TAG      SA      A2,  TEMP
4.    CAT           TLEM    X3,  RATE
5.                  J       MAN
6.    $(3),MAN      ANX     X3,5,S2
```

Explanation:

Lines 1 and 2 are assembled under location counter 0 if no location counter was specified prior to this point.

Lines 3, 4, and 5 are assembled under location counter 2.

Line 6 and subsequent coding are assembled under location counter 3.

1.3.8. Location Counter Reference

Reflexive addressing may be achieved within a symbolic line of coding. The current location counter or a specific location counter can be referenced. The symbol for the current location counter reference is $. When the assembler encounters the $, it inserts the value of the current location counter.

When $+n is placed in the operand field, n-1 lines of code under control of the current location counter are skipped. If the $+n construction is used, care should be taken so that the +n does not extend into a procedure (see Section 3). Extension into a procedure causes particular problems, especially when a variable number of lines of code are generated by the procedure.

Example:

```
        J       $+2
        LA      16,TEMP
        SA      16,WS
```

Explanation:

In this example J $+2 in the first line transfers control to the SA line.

1.3.9. Setting Location Counters

If it were desired to have the object program loaded at relative main storage address 010000, the line

```
$(1)    RES     010000-$
```

should precede source code instruction. The –$ is absolutely necessary if it is not the first line in the program to ensure that any previous value of location counter 1 will be negated.

1.3.10. Line Control

The label field of a line may or may not be blank. The assembler stops interpreting operand information on the basis of one of the following four events, whichever occurs first:

(1) the maximum number of subfields required by the operation has been encountered;
(2) the maximum number of lists required by a PROC reference has been encountered (see 3.1.4);
(3) the 80th character has been read; or
(4) the line terminator (b.b) is encountered.

1.3.11. Line Continuation

If a semicolon (;) is encountered outside of an alphabetic item (see 1.5.4), the current line is continued with the first nonblank on the following line. Any characters on the line after the semicolon are not considered pertinent to the program being assembled, and are transferred to the output listing as comments. A semicolon is used within a comment in order to continue that comment on the next line. If a line is broken within a subfield, the next character must begin in column 1 of the next line.

Example:

```
      1    LABEL     Δ     OPERATION     Δ     OPERAND
1.      LNA     A4,   TABL,3,2
2.        LN;
3.    A4,     TABL;
4.    3,2 .
          . A COMMENT LINE;
MAY ALSO BE CONTINUED ON THE LINE BELOW
```

Explanation:

Lines 2, 3, and 4 produce the same result as line 1.

1.3.12. Comments

The construction, space period space (b.b), terminates a line of coding. Any additional subfields implied by the operation field are taken to be zero. A continuation or termination mark may occur anywhere on a line. Any characters may be entered as comments except the apostrophe (') (see 1.5.4).

Example:

```
 LABEL   Δ   OPERATION   Δ   OPERAND

 LA    A10,   GCART  .  INITIALIZATION;
OF CONTROL FILE SUBROUTINE
```

1.3.13. Ejection of Paper

A slash (/) appearing in column 1 advances paper in the printer to the top of the next page. This same line may also contain a line of coding with the label field starting in column 2. The slash (/) prints on the new page.

1.4. DATA WORD GENERATION

A + or - in the operand field, followed by one to six subfields, generates a constant word. the + or - sign may be separated from the subfields by any number of blanks. If the + sign is omitted, a positive value is assumed. Subfields are separated by commas, which may be followed by one or more blanks.

If the operand field contains one subfield, the value of the subfield is right-justified in a signed 36-bit word unless the value is double precision in which case it is right-justified in two 36-bit words. If the operand field contains two subfields, a data word containing two 18-bit subfields is created; the value of each subfield is right-justified in its respective field. Similarly, three subfields generate three 12-bit fields, four subfields generate four 9-bit fields, and six subfields generate six 6-bit fields. Each subfield in the operand field may be signed independently. If the subfield is preceded by a minus sign, the operand is complemented.

```
 -16384              .  PRODUCES  OCTAL  777777737777
 +'8',-0257          .  PRODUCES  OCTAL  000070777520
 -56,0407,-313       .  PRODUCES  OCTAL  770704077306
  8,-04,21,-28,017,-14
.  THE LINE ABOVE PRODUCES  OCTAL  107325431761
```

If the operand field contains one subfield immediately followed by a D or a value greater than 36 bits in length, the assembler generates a 72-bit value contained in two consecutive 36-bit computer words. The 72-bit value is signed and right-justified.

```
 -16384D        OCTAL  777777777777  777777737777
 +012345543216776.  OCTAL   000000000012  345543216776
```

1.5. EXPRESSIONS

An expression is an elementary item or a series of elementary items connected by operators. Blanks are not permitted within an expression. The combination of single and double precision values generally results in a double precision value (see Appendix D).

1.5.1. Elementary Items

An elementary item is the smallest element of assembler code that can stand alone; an elementary item does not contain an operator.

1.5.2. Octal Values

An octal value may be an elementary item. Such an item is a group of octal integers preceded by a zero. The assembler creates a binary equivalent of the item's value right-justified in a signed field. If the sign is omitted, the value is assumed to be positive.

For example,

```
+017        PRODUCES OCTAL WORD 000000000017
-074        PRODUCES OCTAL WORD 777777777703
-021        PRODUCES OCTAL WORD 777777777756
```

A double precision octal value is produced by writing a constant larger than 36 bits or by placing a letter D immediately after the last octal digit.

1.5.3. Decimal Values

A decimal value may appear as an elementary item within an expression. A decimal item is a group of decimal integers *not* preceded by a zero (see 1.5.2). Such a decimal value, is represented by a right-justified and signed binary equivalent within the object field. If the sign is omitted, the value is assumed to be positive.

For example,

```
+ 12        PRODUCES OCTAL WORD 000000000014
+2048       PRODUCES OCTAL WORD 000000004000
-4162       PRODUCES OCVAL WORD 777777773625
```

A double precision decimal value is produced by writing a value larger than 36 bits or by placing the letter D immediately following the last decimal digit.

1.5.4. Alphabetic Items

Alphabetic characters may be represented in 6-bit Fieldata code as an elementary item. The characters must be enclosed in apostrophes. It is not permissible to code an apostrophe within an alphabetic item. An alphabetic item appears left-justified within its field. If there are less than six characters, the alphabetic item is followed by Fieldata blanks (05 for each blank).

If an alphabetic item is preceded by a plus or minus sign, it may contain a maximum of 12 characters. A positive signed value appears right-justified within its field with the remaining field filled in with zeros. A minus sign preceding the value produces the complement of the value and appears left-justified in the field. If the number of characters is less than seven, only one computer word is used. An alphabetic item used as a literal is assumed to be preceded by a plus sign. A D immediately following the right apostrophe forces double precision (see 1.5.6).

```
 'HEAD'          PRODUCES OCTAL LEFT-JUSTIFIED   151206110505
+'HEAD'          PRODUCES OCTAL RIGHT-JUSTIFIED  000015120611
 'HEAD7890'      PRODUCES  151206116770 716005050505
+'HEAD7890'      PRODUCES  000000001512 061167707160
+'HEAD'D         PRODUCES  000000000000 000015120611
```

1.5.5. Line Items

A line item is any symbolic line, less label, enclosed in parentheses. Line items may be elementary items or expressions.

A literal is represented as an expression enclosed within parentheses and without connecting operators. The assembler then generates a word containing the constant, and this word appears in a literal list at the end of the program. The value of the line item is the address of the generated constant.

Duplicate literals do not appear in the literal list. When location counters are used, the literals appear at the end of the coding associated with a particular counter with only duplicated literals for that particular counter eliminated.

Literals may be double precision if the symbolic line is a single subfield datum of the double precision form. The value of this expression is the address of the first word of the literal.

If the symbolic line of a line item is a data word, the leading + may be omitted. If the symbolic line is an instruction word, the J field may not be supplied in the operation field. Line items within line items are permitted up to eight levels.

Example:

```
    1 LABEL  Δ  OPERATION  Δ  OPERAND
1.  LA       A4,   (5+1)
2.  LA       A4,   +(6)
3.  LA       A4,   (0400)
4.  TE       A5,   (`BRING´)
5.  TNE      A6,   (J PBR)
6.  LA       A8,   ((899))
7.  A        A2,   ((TE A4,(3)))
8.  DL       A4,   (0400)
9.  +(6*3+8/4-2)
```

Explanation:

Line 1. (U) is the address of the literal 6.
Line 2. U address is 6. + (6) is a constant.
Line 3. (0400) is the constant of octal 400. It is a literal.
Line 4. Alphabetic BRING is generated. It is a literal.
Line 5. The instruction J PRR is generated. It is a literal.
Line 6. This instruction will load the location of the location of constant 899 into A7.
Line 7. Two literals will be generated.
Line 8. The constant of 000000000400 will be generated.
Line 9. Not a literal. The line item has a value of 18.

1.5.6. Floating Point and Double Precision

A floating-point decimal or octal value may be represented as an elementary item by including a decimal point within the desired value. The decimal point must be preceded and followed by at least one digit. The letter D must immediately follow the last digit with no intervening spaces. If the sign is omitted, the value is assumed to be positive.

```
+16384.0      PRODUCES FLOATING-POINT WORD 217400000000
±16384.0D     PRODUCES 201740000000 000000000000
    19.0D     PRODUCES 230000000000 000000000000
```

Table 1-2 gives the rules for handling single and double precision expressions.

OPERATION	FIRST VALUE	SECOND VALUE	RESULT
>, <, =	Single	Single	Single
		Double	
	Double	Single	
		Double	
+, −, ++, −−, **	Single	Single	Single (1)
		Double	Double
	Double	Single	
		Double	
*, /, //	Single	Single	Single (1)
		Double	Double (2)
	Double	Single	
		Double	
*/, * +, * −	Single	Single	Single
		Double	
	Double	Single	Double
		Double	

NOTES:

① Multiplication, addition, or subtraction in fixed-point modes may result in a double precision value.
② These cases are not permitted for fixed-point values. If fixed-point values are used, they result in a single precision result with an E error flage.

Table 1-2. Handling of Single and Double Precision Floating-Point Values.

1.5.7. Operators

There are 14 operators in the assembler which designate the method, and implicitly the sequence, to be employed in combining elementary items or expressions within a subfield. Blanks are not permitted within an expression. Evaluation of an expression begins with the substitution of values for each element. The operations are then performed from left to right in hierarchical order as listed in Table 1-3.

The operation with the highest hierarchy number is performed first; operations with the same hierarchy number are performed from left to right. To alter this order, parentheses may be employed but care should be taken to avoid redundant parentheses which may result in the generation of a literal.

If an elementary item or an expression is enclosed in parentheses and an operator appears adjacent to the parentheses, the function of the parentheses in this instance is that of algebraic grouping. The value of this quantity is the algebraic solution of the items or expression enclosed in parentheses. This value should not be confused with the value produced by a literal and therefore is not an address.

All the following operators are assembly-time operators.

HIERARCHY	OPERATOR	DESCRIPTION
Highest 6	*+ *– */	a* + b is equivalent to $a*10^b$ a* – b is equivalent to $a*10^{-b}$ a*/b is equivalent to $a*2^b$
5	* / //	arithmetic product arithmetic quotient covered quotient (a//b is equivalent to $\binom{a+b-1}{b}$)
4	+ –	arithmetic sum arithmetic difference
3	**	logical product (AND)
2	++ ––	logical sum (OR) logical difference (EXCLUSIVE OR)
Lowest 1	= > <	a = b has the value 1 if true, 0 if otherwise a > b has the value 1 if true, 0 if otherwise a < b has the value 1 if true, 0 if otherwise

Table 1-3. Hierarchy of Operators

Table 1-4 gives the rules for determining the arithmetic result of a floating-point operation.

LEVEL	1st ITEM	OP	2nd ITEM	RESULT
6	Any	*+	Binary①	Positive Decimal Exponentiation
	Any	*–	Binary①	Negative Decimal Exponentiation
	Any	*/	Positive Binary①	Positive Binary Exponentiation
	Any	*/	Negative Binary①	Negative Binary Exponentiation Sign filled
5	Any	*	Any	Arithmetic product
	Any	/	Any	Arithmetic quotient
	Any	//	Any	Arithmetic covered quotient
4	Any	+	Any	Arithmetic sum
	Any	–	Any	Arithmetic difference
3	Any	**	Any	Logical product
2	Any	++	Any	Logical sum
		––	Any	Logical difference
1	Any	<,=,>	Any	1 if true 0 if false

NOTE:
① A nonbinary, that is, floating-point value results in an expression error flag (E).

Table 1-4. Arithmetic Results of a Floating-Point Operation

Table 1-5 gives the rules for determining the mode of the result of a floating-point operation.

LEVEL	1st ITEM	OP	2nd ITEM	RESULT
6	Any	*+,*–	Binary①	Floating
	Any	*/	Binary①	Binary
5	Binary	*,/,//	Binary	Binary
	Floating	*,/,//	Binary	Floating
	Binary	*,/,//	Floating	Floating
	Floating	*,/,//	Floating	Floating
4	Binary	+,–	Binary	Binary
	Floating	+,–	Binary	Floating
	Binary	+,–	Floating	Floating
	Floating	+,–	Floating	Floating
3	Any	**	Any	Binary
2	Any	++,––	Any	Binary
1	Any	<,=,>	Any	Binary

NOTE:
① A nonbinary, that is, floating-point value results in an expression error flag (E).

Table 1-5. Mode of Result of Floating-Point Operation

Table 1-6 gives the rules for determining whether the result of a floating-point operation is relocatable.

LEVEL	1st ITEM	OP	2nd ITEM	RESULT	NOTE
1	Any	<, =, >	Any	Not relocatable	
2	Any	++, --	Any	Not relocatable	3
3	Any	**	Any	Not relocatable	3
4	Not relocatable	+, -	Not relocatable	Not relocatable	
	Relocatable	+, -	Not relocatable	Relocatable	
	Not relocatable	+, -	Relocatable	Relocatable	
	Relocatable	+, -	Relocatable	Relocatable	2
5	Any	*, /, //	Any	Not relocatable	3, 4
6	Any	*+, *-, */	Binary	Not relocatable	3, 5

NOTES:

① Floating-point items are never relocatable.

② The difference between two relocatable quantities under the same location counter is not relocatable.

③ Except as noted in ④ , the relocation error flat (R) is set for these operations.

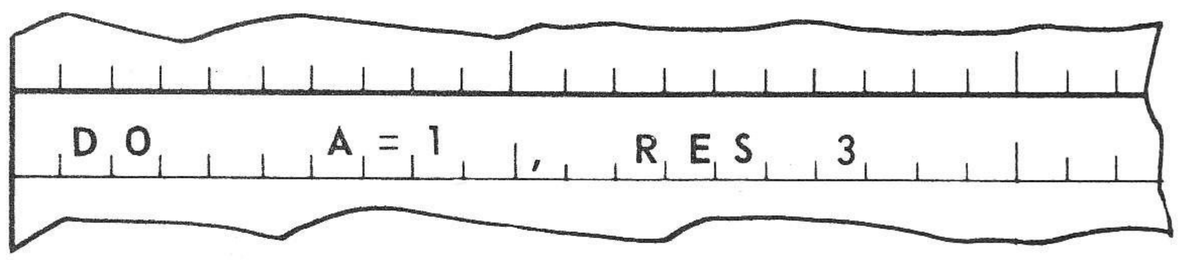
④ Multiplication of a relocatable quantity by an absolute 1, or absolute 1 by a relocatable quantity is relocatable. Multiplication by absolute 0 is absolute 0. In either case, no error flat is set.

⑤ A nonbinary, that is, floating-point value for the 2nd item results in an expression error flag (E).

Table 1-6. Rules for Determining if Results of Floating-Point Operations are Relocatable

1.5.7.1. = Equal

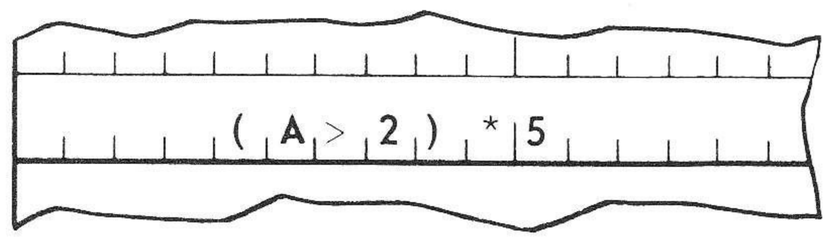
The equal operator compares the values of two expressions. If the two values are equal, the assembler assigns a value of 1 to replace the expression. If the values are not equal, the value of the expression is 0. If A equals 1, the value of the expression is 1; if A is not equal to 1, the value of the expression is 0.



If the condition specified is met (A = 1), the controlling location counter is incremented by 3; otherwise the line is skipped.

1.5.7.2. > Greater Than

The greater than operator compares two expressions. If the value of the first expression is greater than the value of the second expression, the assembler gives a value of 1 to the expression. If the value of the first is equal to or less than the second, a value of 0 is assigned to the expression. If A is greater than 2, the value of the expression is 1; if A is not greater than 2, the value of the expression is 0.



If A is greater than 2, the value of the expression is 5, otherwise the value is 0.

1.5.7.3. < Less Than

The less than operator compares two expressions. If the condition specified is met, a value of 1 results. If the condition is unsatisfied, a value of 0 is assigned to the expression. If A is less than 1, the value of the expression is 1; if A is not less than 1, the value of the expression is 0.

```
A < 2 * 2
```

If A is less than 4, the value of the expression is 1; otherwise the value is 0.

1.5.7.4. ++ Logical Sum

The logical sum operator (OR) provides the logical sum of the values of two expressions.

```
A     EQU   4
      A++14
```

The value of the expression above is 14.

1.5.7.5. -- Logical Difference

The logical difference operator (XOR) produces the logical difference between the values of two expressions.

```
A     EQU   3
      A--18
```

The value of the above expression is 17.

1.5.7.6. ** Logical Product

The logical product operator (AND) produces the logical product of the values of two expressions.

```
N     EQU   17
      +N**3
```

The value of the expression above is 1.

1.5.7.7. + Arithmetic Sum

The arithmetic sum operator produces the algebraic sum of the values of two expressions.

```
LA      A6,     BAR+2
```

Arithmetic register A6 is loaded with the contents of the second word following the word labeled BAR.

1.5.7.8. - Arithmetic Difference

The arithmetic difference operator produces the algebraic difference between the values of two expressions.

```
LA      A11,    KLEE-1
```

Arithmetic register A11 is loaded with the contents of the word immediately preceding the word labeled KLEE.

1.5.7.9. * Arithmetic Product

The value of the first expression is used as the multiplicand; and value of the second is used as the multiplier; the product is obtained by the multiplication of the two expressions.

```
N       EQU     17
        4*4++17
```

The value of the expression above is 17. The 4*4 is operated upon first. The logical sum is then computed.

1.5.7.10. / Arithmetic Quotient

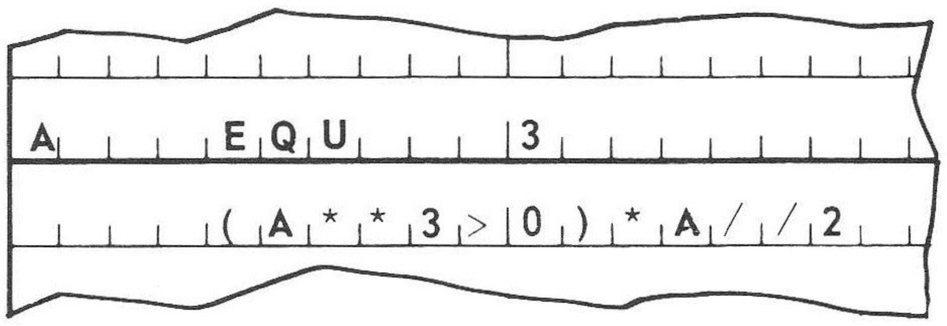
The value of the first expression is the dividend; the value of the second is the divisor; the result of the operation is the quotient; the remainder is discarded by the assembler.

```
Y       EQU     4
        10--8/Y
```

The value of the expression above is 8. The expression 8/Y is operated upon first. The logical difference is then computed.

1.5.7.11. // Covered Quotient

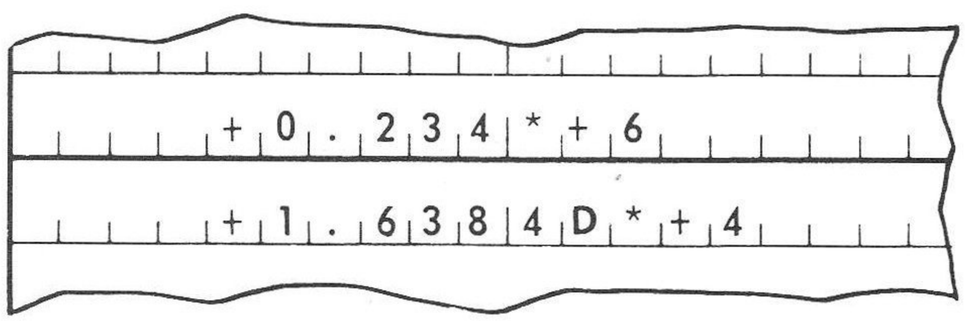
The covered quotient operates in the same fashion as the arithmetic quotient with one difference: if a remainder greater than zero is created during the division, the quotient is increased by one.



The value of the expression above is 2.

1.5.7.12. *+ Positive Decimal Exponent

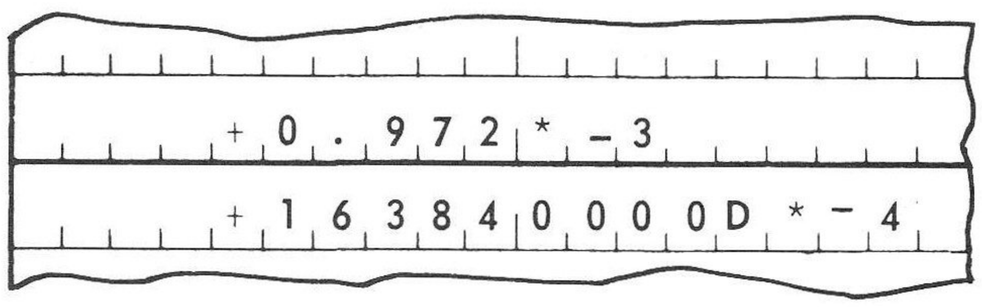
The positive decimal exponent is a method of symbolically creating a floating-point constant in UNIVAC 1106/1108 format. x*+b is equivalent to $x*10^b$.



The value of the first expression above is octal 222711017776. The value of the second expression is 201740000000 000000000000.

1.5.7.13. *– Negative Decimal Exponent

The negative decimal exponent functions in the same manner as the positive exponent. It produces a floating-point constant in UNIVAC 1106/1108 format. x*–b is equivalent to $x*10^{-b}$.



The octal value of the first expression above is 166775467206. The value of the second expression is 201740000000 000000000000.

1.5.7.14. */ Shift Exponent

The shift exponent allows the programmer to enter a number and specify its binary positioning to the assembler. The shift may be left or right according to the sign of the exponent (–b will produce a right shift). x*/b is equivalent to $x*2^b$.

```
1  LABEL     Δ     OPERATION     Δ     OPERAND     Δ     COMMEN
Z  DO  72,   +0400000000000000000000000000000D  */(1-Z)
```

The line above will generate 144 words.
The octal value of the first pair of words is 400000000000 000000000000.
The octal values of the final words are 777777777777 777777777777.

2. ASSEMBLER DIRECTIVES

2.1. DIRECTIVES – GENERALIZED FORMAT

The symbolic assembler directives control or direct the assembly processor just as operation codes control or direct the central computer processor. The assembler directives are represented by mnemonics written in the operation field of a symbolic line of code. The flexibility of the directives is the key to the power of the assembler. The directives are used to equate expressions and to adjust the location counter value, and allow special controls over the generation of object coding. The general format for directives is:

LABEL DIRECTIVE EXPRESSION

Not every directive uses all three fields. Symbols appearing in the operand field of directives must be defined prior to their appearance in the directive. A detailed description of each of the 15 assembler directives follows.

2.1.1. The Equate Directive, EQU

The EQU directive equates a label appearing in its label field to the value of the expression in the operand field. The EQU directive must include all three fields:

LABEL EQU EXPRESSION

A value defined by the EQU directive may be referenced in any succeeding line of coding by using the label equated to it. If a label is to be assigned a value, it must appear in an EQU line. If the label is not equated to a value prior to its use or reference, the label is considered undefined. Labels equated to relocatable expressions may appear anywhere in the program.

It is possible to generate a double precision equate statement by having the operand contain one numeric subfield immediately followed by the letter D with no intervening spaces.

If a particular expression is used frequently throughout a program or procedure, it is highly expeditious to use the EQU directive to substitute a simple label for the entire expression.

Example:

```
A       EQU     7/22816*31+(5//B)
        LA      16,2*A
C       EQU     0646473621012345D
        +       C
```

Labels defined by an EQU directive are relocatable only if the value is relocatable.

```
1. CONST     EQU     010000
2.       LA      A4,CONST
```

The U portion of the instruction generated on line 2 becomes 010000. As a result, the LA instruction loads arithmetic register 4 with the contents of storage location 010000.

```
1. L         EQU     010000
2. A4        EQU     16
3.       LA      A4,L
```

L is equated to the value 010000. Line 2 equates A4 to 16. The third line produces object code 10 00 04 00 0 010000.

2.1.2. EQUF Directive

If repeated reference is made to the j and u fields of an instruction word, it may be desirable to have one symbol represent these fields.

Example:

```
1. A         EQU     +(I$,0,J,0,0,0,U)
2. SA            A1,     A
```

I$ refers to the field form: 6,4,4,4,2,16.

SA, the mnemonic for the Store A Register instruction, refers to the I$ form. The value of the A is the line item in the operand field of the EQU directive. For j = 014 and U = 010164, the result is:

```
01  00  01  00  0  000000    from SA
00  14  00  00  0  010164    line item
-------------------------
01  14  01  00  0  010164    output word
```

The EQUF directive builds a nonliteral line item which includes the u, x, j, and h-i subfields in the I$ form. The format is:

LABEL EQUF expressions

Example:

```
FIELD EQUF  0100,8,14
```

The three operand expressions represent u, x, and j, respectively. The field definition thus created could be referred to in this manner:

```
    LA  16,FIELD
```

giving

10	00	04	00	0	000000	from LA
00	16	04	10	3	000100	from line item in EQUF
10	16	04	10	3	000100	output word

Another example:

```
1.  FSTR    EQU     20
2.  FREGS   EQUF    FSTR+12,X4
3.  SA      A5,     FREGS
```

Line 2 identifies the FSTR as appearing in index register 4. The field definition appears as:

01	00	05	00	0	000000	from Line 3
00	00	00	04	0	000040	from Line 2
01	00	05	04	0	000040	output word

Care should be taken so that nonzero fields are not ORed (see 2.1.4.1).

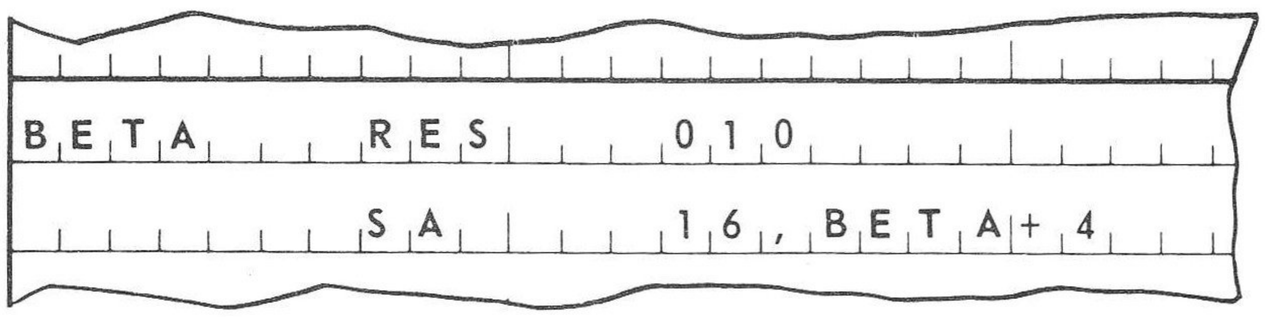
2.1.3. The Reserve Directive, RES

The RES directive increments or decrements a control counter. The operand field of the directive contains a signed value e that specifies the desired increment if positive, or decrement if negative. This value may be represented by an expression. The format is:

LABEL RES e

Symbols appearing in the expression e must be defined prior to the RES line in which they appear.

The RES directive may be used either to create a work area for data, which is not cleared to zeroes, or to specify absolute location counter positioning to the assembler. If a label is placed on the coding line which contains a RES directive, the label is equated to the present value of the control counter which is the address of the first reserved word.



The line above will store the contents of register A4 into the fifth word of the reserved area BETA.

2.1.4. The Format Directive, FORM

The FORM directive describes a special word format designed by the user. The word format may include fields of variable length. The length in bits of each field is defined in the operand field of the FORM directive. The value entered in the operand field specifies the number of bits desired in each field. The format is:

LABEL FORM $e_1,e_2,\ldots,e_n$

The number of bits specified by the sum of the values of the operand expressions must equal 36 or 72 depending on whether a single or double precision form word is desired.

By writing the label of the FORM directive, the form defined in that line of coding may be referenced from another part of the program. The label of the FORM line is written in the operation field and is followed by a series of expressions in the operand field. The expressions in the operand field specify the value to be inserted in each field of the generated word or words.

A reference to a specific FORM label always creates one or two words composed in the format specified. Truncation occurs and an error flag is set if a given value exceeds the space indicated in the associated field in the FORM directive.

The series of expressions in the operation field must equal the number of fields specified on the FORM directive. If the number of fields is not equal, an E flag is set.

```
INSTR  FORM   6,4,4,4,2,16
       INSTR  054,0,04,01,0,010002
```

The line above would produce the following:

54 00 04 01 0 010002 (assembler listing)
54 01 01 01 2 010002 (in main storage)

In the example below a truncation flag is set.

```
INSTRA FORM   6,18,12,18,6,6,6
       INSTRA 027,01000,10,1000,167,012,4
```

The line above would produce the two words as follows, plus an error flag because the expression 167 (247_8) requires more than six bits.

270010000012 001750471204 (in main storage)

2.1.4.1. ORing of Forms

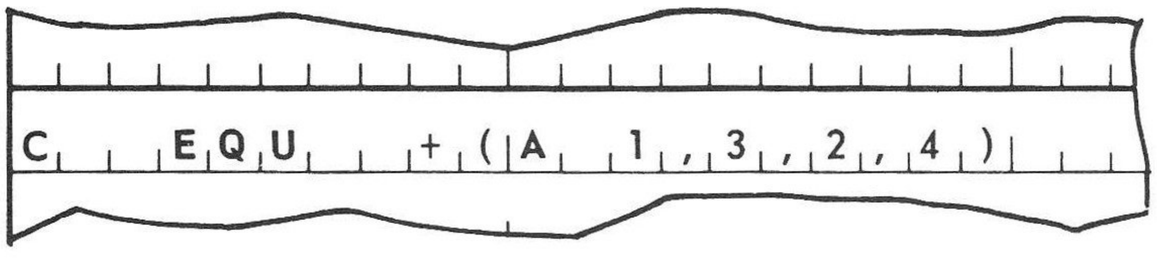
A field in a FORM reference line can be a line item. If the form of the line item is identical to the form referenced, and is not a literal, the corresponding fields from borh form references are ORed.

For example, consider two identical forms:

```
A      FORM   9,9,9,9
B      FORM   9,9,9,9
```

The four fields of A and B are 9 bits in length.

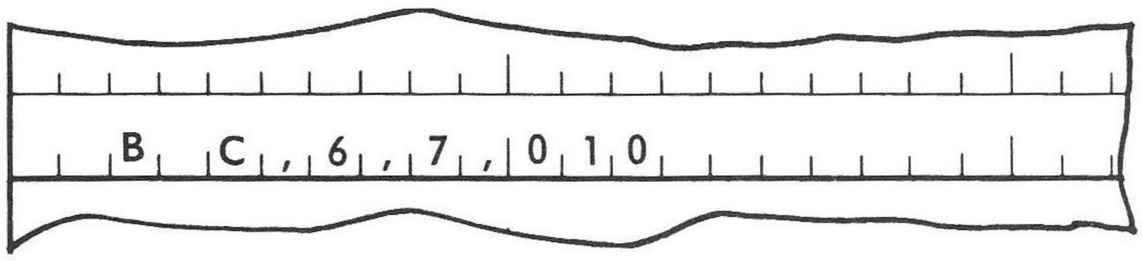
A line item is created with one of these forms and its value is equated to a label:



C shown in octal is:

001 003 002 004

The + preceding the line item suppresses creation of a literal. Next, using C as one of the values, refer to form B. If two forms have nonzero values in any common position, those fields are ORed.



The results are shown in octal:

C = 001 003 002 004
B = 000 006 007 010

result = 001 007 007 014

The nonzero bit positions have been ORed. Field 1 of B is all zeros as C has taken that position.

2.1.5. The END Directive, END

The processing of an END directive indicates to the assembler that it has reached the end of a logical sequence of coding. The format is:

```
        END     e
```

An END line must not include a label. The operand field (e) is used only with functions (see 3.4).

The interpretation of the operand of an END directive depends on its associated directive. When an END directive terminates a program assembly, the operand field specifies the starting address in the object code produced at execution time.

2.1.6. The Literal Directive, LIT

The LIT directive defines a literal table under the control of the location counter in use when this directive is encountered by the assembler. The format is

LABEL LIT

with no operand. The label is optional.

Through the use of LIT directives, a number of separate literal tables can be created. Duplicate literals are eliminated within each unique literal lable; however, duplicates may exist in separate literal tables. In the absence of a LIT directive, all literals are placed in the literal table under control of location counter 0.

```
LA   16,(04)
```

In the example above, the octal literal 000000000004 is placed in the literal table controlled by location counter 0 because no LIT directive has been used.

If a literal table not under control of location counter 0 is required, a LIT directive must be used. A specific location counter is declared by writing $ (e) starting in column 1, the e being any location counter number 0 through 31.

```
$(2)  LIT
      LA   16,(04)
```

The octal literal 000000000004 is placed in the unlabeled literal table controlled by location counter 2.

Labels may follow the declaration of a specific location counter. The format is:

$(e),LABEL

There may be no intervening spaces. The label may not be subscripted or suffixed by an asterisk nor may the label be referenced.

If all LIT directives in a program have labels, any literal not preceded by a label is placed in the literal table under the control of location counter 0.

If a LIT directive has a label, all literals to be placed in this literal table must be preceded by the label associated with this LIT directive.

```
$(2),TOM   LIT
           LA     16,(04)
           LX     3,TOM(01000)
```

The literal 000000000004 is placed in the unlabeled literal table controlled by location counter 0. The literal 000000001000 is placed in the labeled literal table controlled by location counter 2.

If the LIT directive has no label, all subsequent literals not preceded by a label are placed in the literal table designated by the LIT directive until another unlabeled LIT directive is encountered. Any number of unlabeled LIT directives may appear throughout a program, each having this same effect. If desired, unlabeled literals can follow each program segment for which a separate location counter is used. Only one unlabeled literal table is allowed for each location counter.

Example:

```
1. $(2),TOM
2. $(18)       LIT
3.             LA     A4,     (04)
4.             LXI    X3,     (TEXT')
5. $(9)        LIT
6.             LA     A5,     (100)
7.             LX     X4,     TOM(635)
```

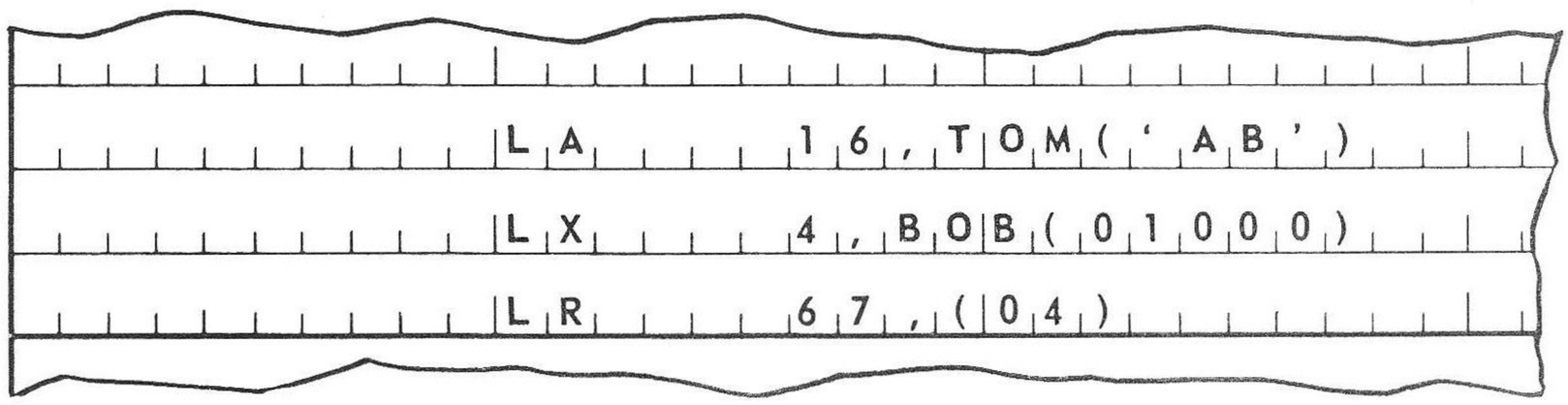
Explanation:

The literals in lines 3 and 4 are placed in the literal table controlled by location counter 18.

The literal in line 6 is placed in the literal table controlled by location counter 9.

The literal in line 7 is placed in the literal table controlled by location counter 2.

Example:



The literal 000000000004 is placed in the table controlled by location counter 0. TOM and BOB were defined by previous LIT directives.

Literals are generated only during pass two of the assembler. Unlabeled literals are generated under location counter 0 until a LIT directive with a blank label supersedes this arbitrary selection of location counters.

2.1.7. The Information Directive, INFO

The format of the INFO directive is:

LABEL INFO a c_1

The label is optional; a represents one of two types of storage that is assigned by the collector. If a is 4, location counter c_1 is assigned blank common. If there is a label and a is 2, location counter c_1 is assigned as labeled common block. For a discussion of common blocks see *UNIVAC 1108 Multi-Processor System FORTRAN V Programmers Reference Manual, UP-4060* (current version).

2.1.8. The DO Directive, DO

The DO directive causes the assembler to process a line of coding a specified number of times. Two entries appear in the operand field of this directive. The second operand entry may be any valid symbolic line with or without a label. The number of times this line is processed is determined by the value of the expression contained in the first operand entry which is called the DO count. The two operand entries are separated by blank comma (b,). If the second operand entry does not have a label, the construction space comma space (b,b) is used between the first and second operand.

The formats are:

LABEL	DO	e_1 b, b LINE OF CODING
LABEL	DO	e_1 b, LABEL b LINE OF CODING

The label is optional. When used, the label is equated to a counter whose initial value is always 0. Each time the line of coding is processed, this counter, the label value, is incremented by 1 until the required limit specified by e_1 is reached. Thus, if the DO count is 0, the final value of the label is 0.

```
K  DO  10  ,  +K
```

The value K is generated ten times. The first value of K is 1; the last value is 10.

If the DO count is negative, the E flag is set (see Appendix C) and no lines are generated.

The DO directive may be conditional. That is, the DO count may depend on alteration of previously assigned values. For example:

```
DO  A<B  ,  -3
```

If A is less than B, the expression A<B is true, and its value is 1. Thus, the line –3 is processed once. If A is not less than B, the expression is false, its value is 0, and the line –3 is not processed.

The first operand, DO count, may be an expression to be evaluated as:

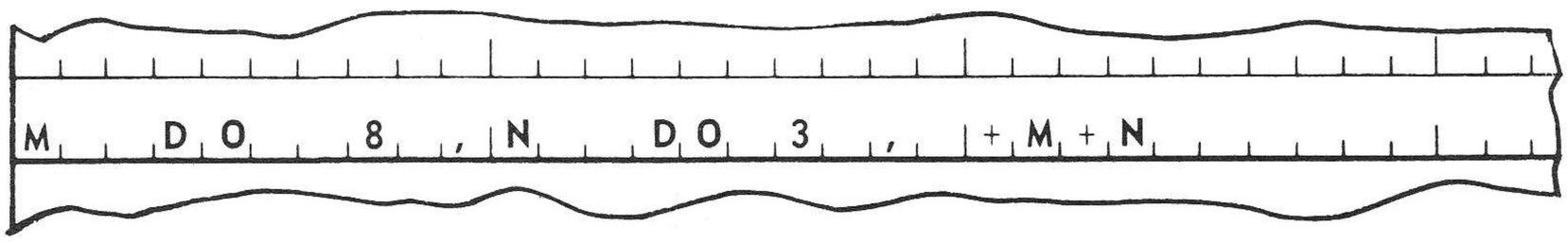
```
A  EQU  6
L  DO  ((A**7)-5)+4  ,  TAG(L)  +L*2
```

In this example the DO count is evaluated as 6–5+4=5. As a result, the data line is processed five times as though it had been coded as follows:

```
TAG(1)    +2
TAG(2)    +4
TAG(3)    +6
TAG(4)    +8
TAG(5)    +10
```

In this case +L*2 is evaluated successively as 1*2, 2*2, 3*2, etc.

DO statements may be nested within DO statements up to eight levels. Execution of statements proceeds from innermost to outermost.



This line produces a table of 24 entries as follows:

```
M = 1, N = 1    value is 2
       = 2    value is 3
       = 3    value is 4
M = 2, N = 1    value is 3
       = 2    value is 4
       = 3    value is 5
       .          .
       .          .
       .          .
M = 8, N = 1    value is 9
       = 2    value is 10
       = 3    value is 11
```

2.1.9. Listing Directives, LIST and UNLIST

These two directives enable the programmer to control the listing of the assembler. The LIST directive negates the effect of no options on the ASM control card or a previously used UNLIST directive which suppressed the listing. They may be used as often as desired in the source code, but must be removed when a complete listing is desired. It should be noted that the image containing the UNLIST directive is not printed and the image containing the LIST directive is printed. The format is:

```
LIST
UNLIST
```

Neither label nor operand are used.

2.2. SPECIAL DIRECTIVES

Three special assembler directives are available to assist in defining an object computer to the assembler. Use of them overrides certain built-in definitions for the 1106/1108 assembler. The directives are:

WRD – Redefines the word length (in bits) for the object machine.

CHAR – Redefines the character set for the object machine.

NEG – Redefines the format of negative values for the object machine.

2.2.1. The Word Directive, WRD

The WRD directive indicates the object computer word size in bits. When an output word is generated, it must not exceed the stated output word size, or a truncation error is noted. This limitation is ignored during the evaluation of expressions, since values are limited only to the double precision word size, 72 bits. Only when a 'line item' is generated will the defined output word size be considered. The format of the WRD directive is:

```
WRD     e
```

where e is any expression with a value equal to or less than 72. For example, WRD 18 indicates an 18-bit word size for this assembly. To illustrate the effect of the directive, symbolic lines are shown side by side with the octal code which would be produced by the assembler. The 1106/1108 character set (Fieldata) is assumed:

Line	Output
'ABCDEFG'	060710 111213 140505
+ 0	000000
+64	000100

2.2.2. The Character Directive, CHAR

The CHAR directive is used to alter translation of the 1106/1108 character set to an alternate set of 6-bit equivalents. The translation takes place any time the assembler encounters one or more characters enclosed by apostrophes. The format is:

CHAR $c_1,e_1,c_2,e_2,\ldots,c_n,e_n$

where, for each pair of expressions, c is the value of the 1106/1108 character to be replaced by e. The value of both the c and e expressions must be in the range 0 to 077. If greater than 077, a T error flag marks the line (see Appendix C).
For example, if

```
CHAR  6,024,7,025,010,026
```

were used, the characters 'A', 'B', and 'C' would be given the values 024, 025, and 026, respectively. Alternately, if

```
IDO 3,CHAR I+5,I+023
```

were used, it would have the same effect. Here are output examples, assuming a 36-bit word length:

Line	Output
'AABBCC'	242425252626
+ 'A'	000000000024
+ 'A', 'B'	000024 000025

2.2.3. The Negative Directive, NEG

The NEG directive is used to alter the output format of negative quantities and has value when assembling for alternate computers with negative representation other than ones complement form. The format is:

LABEL NEG e

where the label has a special interpretation and represents the value of an 1106/1108 negative quantity in the expression e which contains the label which gives in 1106/1108 terms the value to be used as output for negative values. For example, if the following signed magnitude algorithm were used,

DUM NEG DUM--0377777777777

then the following values would be output for the indicated negative quantities:

SOURCE VALUE		OUTPUT VALUE
−5		400000000005
+2, −3	T	000002000003

The T error flag indicates that the minus value cannot be placed in an 18-bit field.

Similiarly,

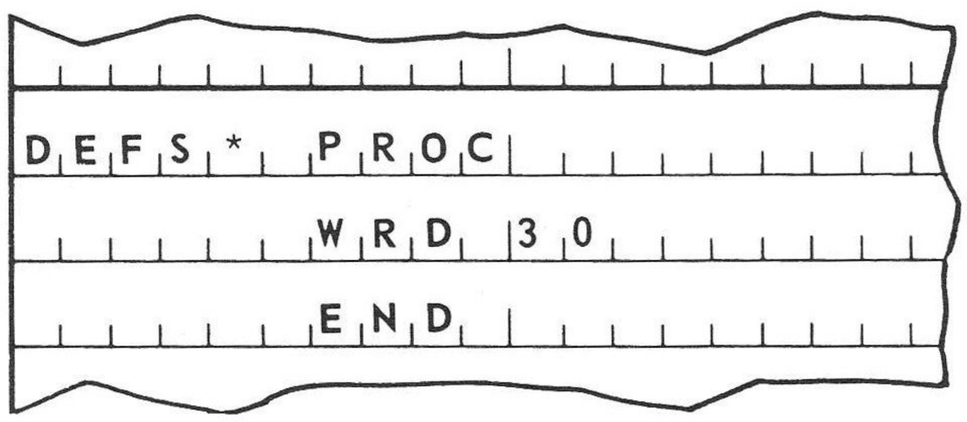
NG NEG NG+1

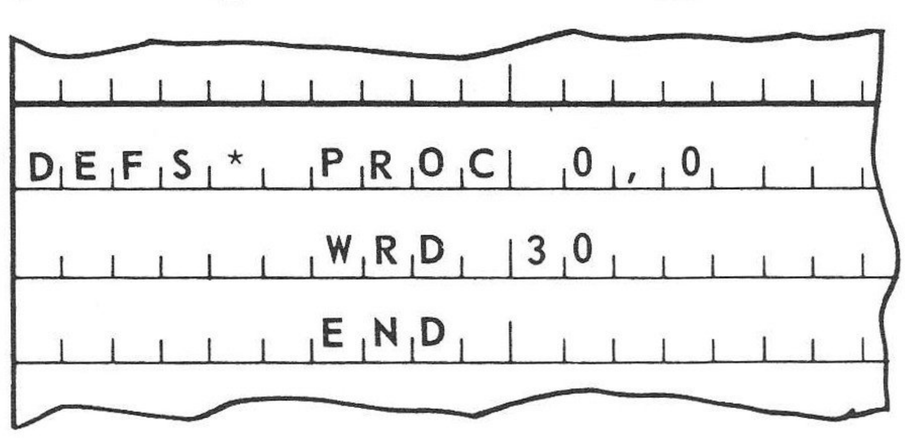
would result in negative values being output in twos complement form.

−14	777777777762

2.2.3.1. Usage of Special Directives

These directives must precede lines of symbolic code which are to be affected by them. If they are coded within a procedure, the procedure must be explicitly referenced by name to get the effect of the special directives. Furthermore, such a procedure must be subassembled in assembly pass 1. Do not code the second PROC directive operand. The first example following is correct; the second is not.





After a special directive is encountered by the assembler, its effect continues until another is encountered. Also, the effect is available at all levels of processing, whether or not in a procedure. For instance,

```
                WRD     30
(1)             etc
    P*          PROC
                WRD     24
(2)             etc
                END
(3)             etc
                p
(4)             etc
(5)             WRD     30
(6)             etc
```

For code at line	Word length, in bits, is
1	30
2	24
3	30
4	24
5	30

The WRD directive in P procedure has no effect outside the procedure until P is referenced. All coding following the reference to P produces 24-bit words. All coding following the WRD redefinition at line 5 produces 30-bit words, until either another reference to P or another redefinition is encountered.

3. PROCEDURES AND FUNCTIONS

3.1. PROCEDURES

Often a program requires repetitive sequences of coding. These sequences are not necessarily identical but there is enough similarity to make the writing of these sequences mechanical. The *procedure* is a method employed by the assembler which permits the automatic generation and modification of repetitive coding sequences. Procedures are implemented by the PROC directive. The PROC directive uses procedure samples to generate the required coding. As the assembler encounters each procedure sample, it stores the procedure and the procedure's entry points. When a call to the procedure is encountered, the assembler references the procedure entry point table, locates the procedure, and then generates the required coding. The procedure sample must physically precede any call to it in the main program.

3.1.1. Sample Procedures

A procedure sample must begin with a PROC directive and end with an END directive. The PROC and END directives are the delineators of the procedure.

3.1.2. PROC Directives

The format of the PROC directive is as follows:

LABEL PROC OPERAND

The label field contains any label not exceeding six characters. The label identifies the specific PROC and is the means by which the procedure is referenced.

The operation field contains the PROC directive. This directive signals the assembler that sample coding is to follow.

The operand field may contain zero, one, or two expressions. Subfield 1 contains a value specifying the maximum number of fields appearing on that procedure's call line (see 3.1.4.1).

Subfield 2 of the operand field cannot be coded unless a value appears in subfield 1. The value entered in subfield 2 indicates the maximum number of lines of code generated when the sample is referenced. Subfield 2 must be omitted in the following situations:

- if forward references are made in the procedure (see 3.1.9);
- if external definitions are made in a procedure (except entry points);
- if the procedure could generate a variable number of lines;
- when a change of location counter control occurs within a procedure; or
- when a label on a procedure reference line is to be assigned to a line other than the first line of the procedure.

Except for the conditions stated above, subfield 2 should be used because it eliminates one assembly pass, thereby shortening assembly time.

A line terminator (b.b) must precede any comments on the PROC directive line.

```
    1 LABEL      Δ OPERATION      Δ OPERAND
1.  COMPAR       PROC 1,10 .
2.  MOVE         PROC .
```

Line 1 contains the label COMPAR. Subfield 1 in the operand specifies one field may appear on the reference line. Subfield 2 indicates a maximum of ten lines will be generated by the procedure.

Line 2 has no operand field. The period halts the scanning of the line thereby reducing assembly time.

3.1.3. END Directive

The END directive must appear at the end of each procedure. END is coded in the operation field. The label and operand fields are left blank.

Example of a simple procedure:

```
1.  LOAD         PROC .
2.               LA   15,TAG
3.               END
```

- Lines number 1 and 3 contain the PROC and END directives which serve as delimiters to the sample contained between them.
- LOAD is the label by which this procedure may be referenced.

Each time this procedure is called, the code provided by line 2 is generated.

3.1.4. Referencing a Procedure

When a procedure reference is encountered at assembly time, code from the specified procedure is generated. The procedure must appear physically in the main program before it is referenced. To reference a procedure, a call line is used.

3.1.4.1. Definition of a Procedure Call Line

A procedure call line informs the assembler that generation and modification of a code sequence are to begin at this point. The operation field contains the external label of the procedure desired. The operand field contains the expressions (parameters) needed for modification. The format of a call line is:

LABEL PROC LABEL OPERAND

The label field of a call line is optional; if used, the label starts in column 1 and may contain from zero to five alphanumeric characters.

The operation field contains the label of the desired procedure.

The operand field contains the parameters needed to modify the procedure.

A period should be used to terminate the call line. This halts the scan of the line and reduces assembly time.

```
      1    LABEL     Δ     OPERATION     Δ     OPERAND
1.         LOAD    .
2.   CALL1         SPEC    .
3.   ADDP          ADD22     4,TAG     99,PUR    .
```

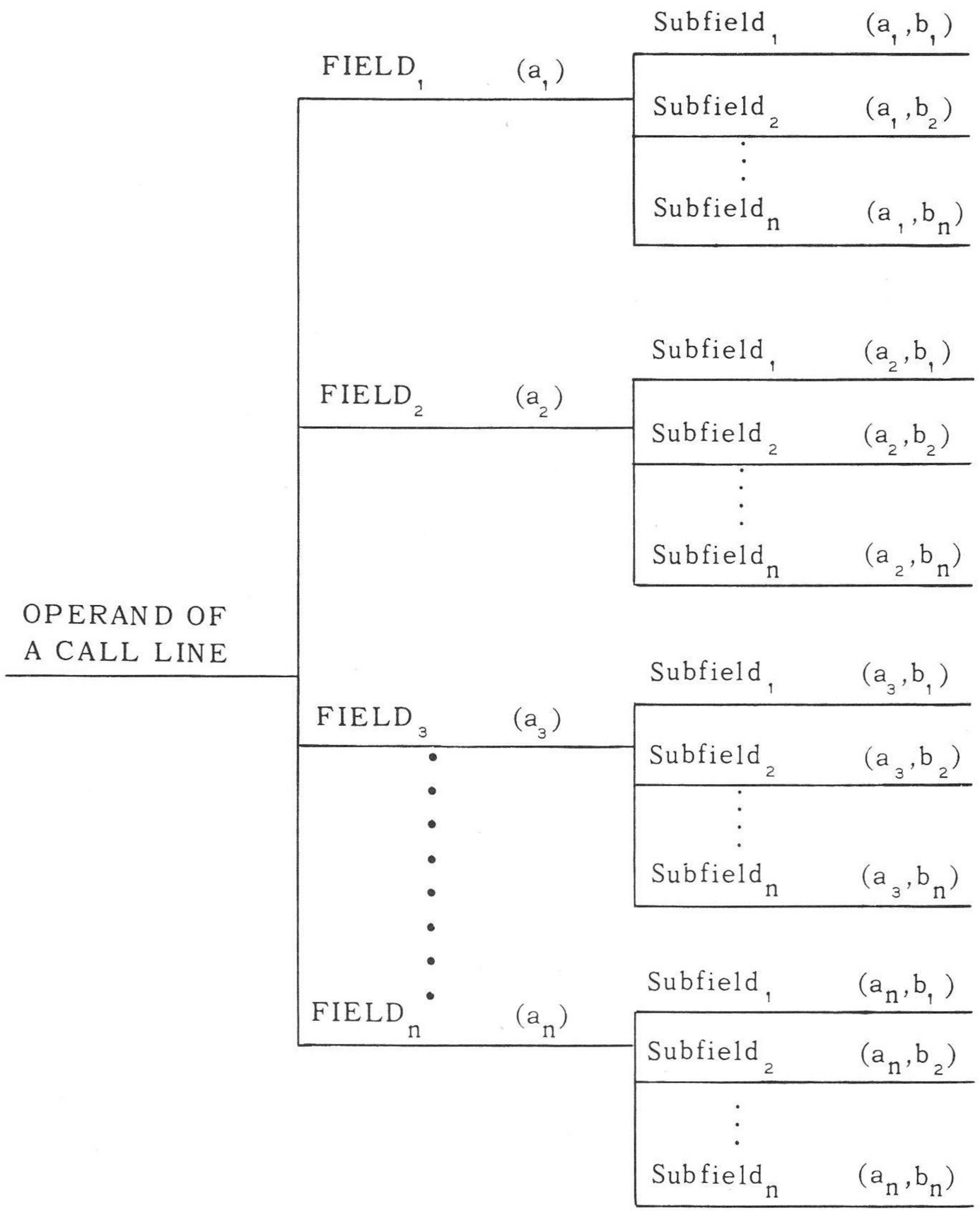
Line 1 has no label. The LOAD procedure will be generated.

Line 2 contains the label, CALL 1. The procedure referenced is SPEC.

Line 3 contains the label ADDP. The procedure ADD22 is referenced. The operand field contains four parameters (see 3.1.4.2).

3.1.4.2. The Operand Field of a Call Line

The operand field of a call line may contain parameters used to modify values appearing within a procedure. These parameters appear in fields and subfields of the operand. There may be any number of fields and any number of subfields may appear within the fields. Fields are separated by blanks; subfields are separated by commas.



Example:

```
1  LABEL  Δ  OPERATION  Δ  OPERAND
   LA     A9,6,4,SLT  JIM,INST  W,R,S,T
```

Spaces separate fields; commas separate subfields.

Explanation:

Field 1 contains subfields 6, 4, SLT.
Field 2 contains subfields JIM, INST.
Field 3 contains subfields W, R, S, T.

3.1.5. Paraforms

The parameter reference form, commonly called the paraform, is a device for selectively obtaining parameters in the operand field of a call line. Paraforms appear in the operand field of a line of symbolic coding and are used only within the bounds of a procedure.

A paraform consists of the name of the procedure referenced, immediately followed by a set of parentheses. Enclosed in the parentheses are two values separated by a comma (a,b). The a refers to a specific field in the call line. The b refers to a specific subfield within the specified (a) field.

Example:

```
      1
      LABEL    Δ    OPERATION    Δ    OPERAND
1.    CALL          MOVE*        TAG,X,Q,ZETA    A1O,BET.
2.                  SQRT*        4,5,12,6        A,B     TAG,TE.
3.    ALPHA         DIV*         3,1,9
```

Explanation:

Line 1 contains the label CALL. There are six parameters appearing in two fields. Field 1 contains four subfields; field 2 has two subfields.

Line 2 has no label. There are eight parameters appearing in three fields. Field 1 has four subfields; field 2 has two subfields; and field 3 has two subfields.

Line 3 has three parameters contained in one field.

The following is a call line for the MOVE procedure:

```
CALL3   MOVE*   TAG,Q   R,35,T.
```

The operand expressions on a procedure call line provide specific values for a general framework of coding.

```
1.  MOVE*    PROC
2.           LA      A3,     MOVE(2,1)
3.           SA      A5,     MOVE(1,1)
4.           AN      A9,     MOVE(1,2)
5.
```

A procedure structure must appear physically in the main program before it is referenced.

Explanation:

Line 1 is the PROC directive. MOVE is the label of the procedure and the means by which the procedure is referenced. The asterisk makes the MOVE PROC available to the other procedures (see 3.1.8).

Line 2. Register A3 is loaded with the parameter appearing in the first subfield of the second field of the call line, in this case, R.

Line 3. Register A5 is loaded with the parameter appearing in the first subfield in First Field, in this case, TAG.

Line 4. The parameter appearing in the second subfield of the first field is subtracted from register A9, in this case, Q.

Paraform constructions are summarized below (PL stands for PROC label):

PL(a)	The value generated by (a) is equal to the number of subfields in the specified (a) field.
PL(a,b)	The value generated by (a,b) is the parameter appearing in the b subfield of field a.
PL	When the procedure label is written with no specified field or subfield, the value generated is a constant equal to the number of fields in the call line. If entry was made by a NAME line, this figure is greater by 1 (see 3.2.2).
PL(a,*b)	Generates a value of 1 if the b subfield of field a is preceded by an asterisk. If there is no asterisk, the value generated is 0.
PL(0,0)	Refers to the operand on a NAME line. It is undefined and set equal to 0 if entry was not made by a NAME directive (see 3.2.2).
PL(0,1)	Refers to the second subfield of the operation field given at time of call in a NAME directive (see 3.2.2). It is undefined if entry is not made by a NAME directive.
PL(0,n)	Refers to the (n+1) subfield of the operation field submitted with a NAME directive (see 3.2.2).

Example:

```
1    LABEL      Δ      OPERATION      Δ      OPERAND
CALL4           ADDP*      TAG,Q          FILE,A,B        *YEL
```

Below are lines of code appearing within a procedure:

```
      1
      LABEL     Δ     OPERATION          Δ
1.      LX        X2,      ADDP
2.      LA        A4,      ADDP(2,1)
3.      LA        A9,      ADDP(1)
4.      LX        X5,      ADDP(3,2)
5.      LX        X9,      ADDP(3,*1)
```

Line 1. The value generated for the operand field is equivalent to the number of fields in the call line. In this case, 3 is the value.

Line 2. The value generated for the operand field appears in the first subfield of the second field of the call line. In this case, FILE is the value generated.

Line 3. The value generated is a constant that is equivalent to the number of subfields contained in the specified field. In this case, the constant 2 is generated (field 1 has two subfields).

Line 4. The value generated is 0 (field 3 does not have two subfields).

Line 5. The value generated is 1 (asterisk precedes the referenced parameter).

In the procedure example shown below, a constant is added to a given value, and the result is stored. With each call, the given value and the storage location vary.

```
ADDP*     PROC
          LA    16,(ADDP(1,1))
          AA    16,CONSTA
          SA    16,ADDP(1,2)
          END
```

The entrance point is indicated by the asterisked label. At each call, the Load A (LA) instruction references the first subfield of the first field in the operand field of the call line. Similarly, the Store A (SA) instruction references the second subfield of the first field. In this example, the first field is the only field.

The following exemplifies a call on this procedure:

```
ADDP   063,RAM
```

This generates three lines of coding, as if they had been written:

```
LA     16,(063)
AA     16,CONSTA
SA     16,RAM
```

A second call on the procedure might be:

```
ADDP   064,RAM2
```

The equivalent coding is:

```
LA     16,(064)
AA     16,CONSTA
SA     16,RAM2
```

3.1.6. Subassembly Technique

The assembler employs a technique which allows it to temporarily discontinue the current assembly and begin a subassembly. Upon completion of the subassembly, the original assembly continues where it left off. It is possible for the subassembly to reference any value of the assembly which caused the subassembly to take place. In turn, the subassembly may generate values or code which can be handed up to the the calling assembly.

The user device which directs the assembler to begin a subassembly is the PROC directive. The source code to be assembled is the sample provided by the user and delimited with the PROC and END directives. Subassemblies may in turn begin other subassemblies; these may be nested down to 62 levels.

Since the assembler employs this technique, the overhead involved with a procedure (that is, interrupting current assembly, sectioning tables, limiting references and definitions, restarting the original assembly) is somewhat costly in terms of time.

3.1.7. Nesting of Procedures

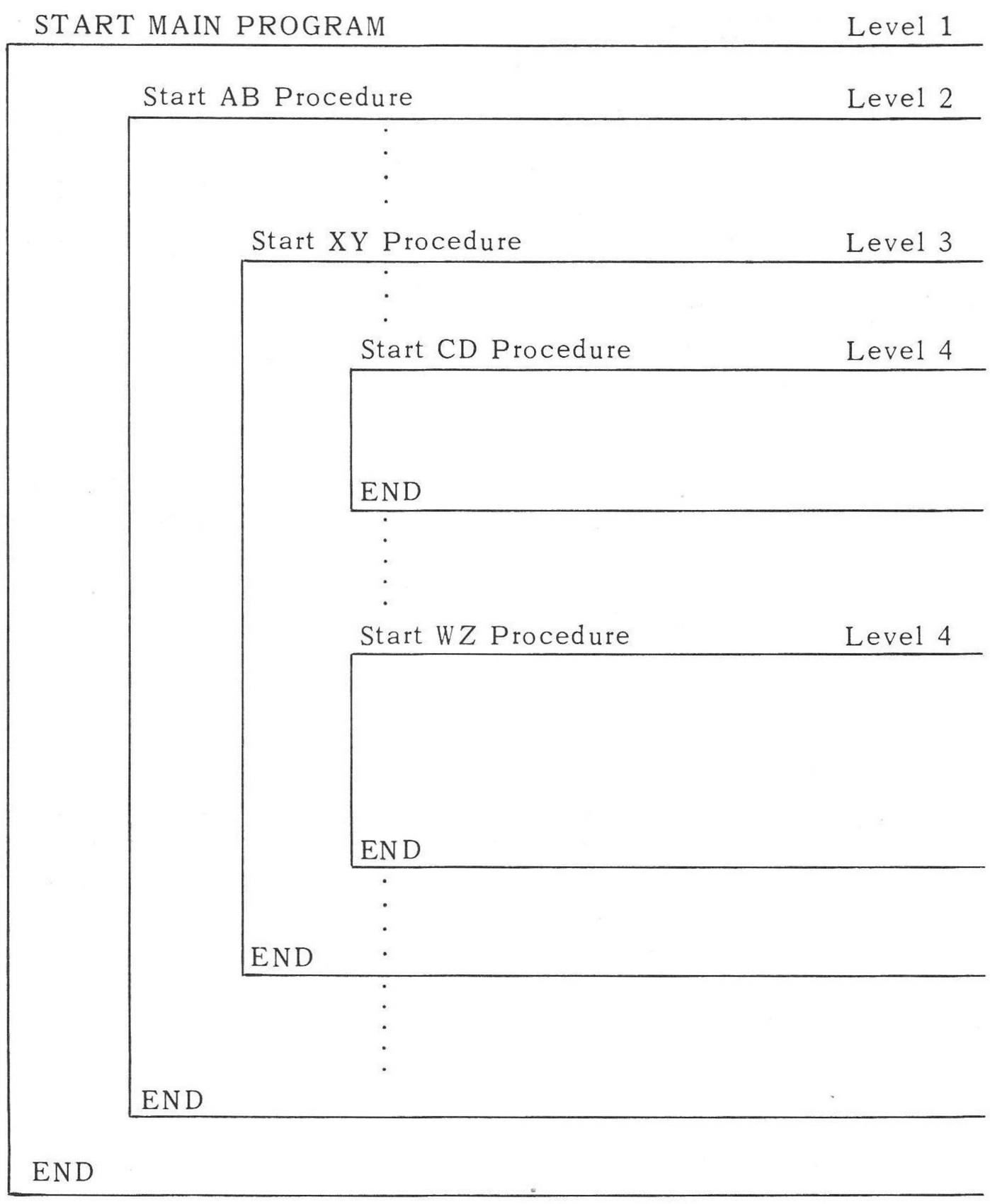
The nesting of procedures enables the programmer to use the same label in different procedures. Therefore, no duplicate (D) flag is generated when the duplicate labels are encountered by the assembler. Nesting allows simpler block building techniques but requires longer assembly time.

Where practical, the depth of nesting should be limited. Use of distributed NAME lines and the GO directives may be helpful in restricting levels of nesting (see 3.2.2. and 3.2.3).

3.1.7.1. Physical Nesting

Physical nesting occurs when a procedure is physically located within the bounds of another procedure. If a procedure is physically contained within another procedure, the internal procedure is considered to be one level lower than the external procedure. Procedures may be nested to 62 levels. An internal procedure may be referenced if a call is made on the external procedure.

Example of Physical Nesting:



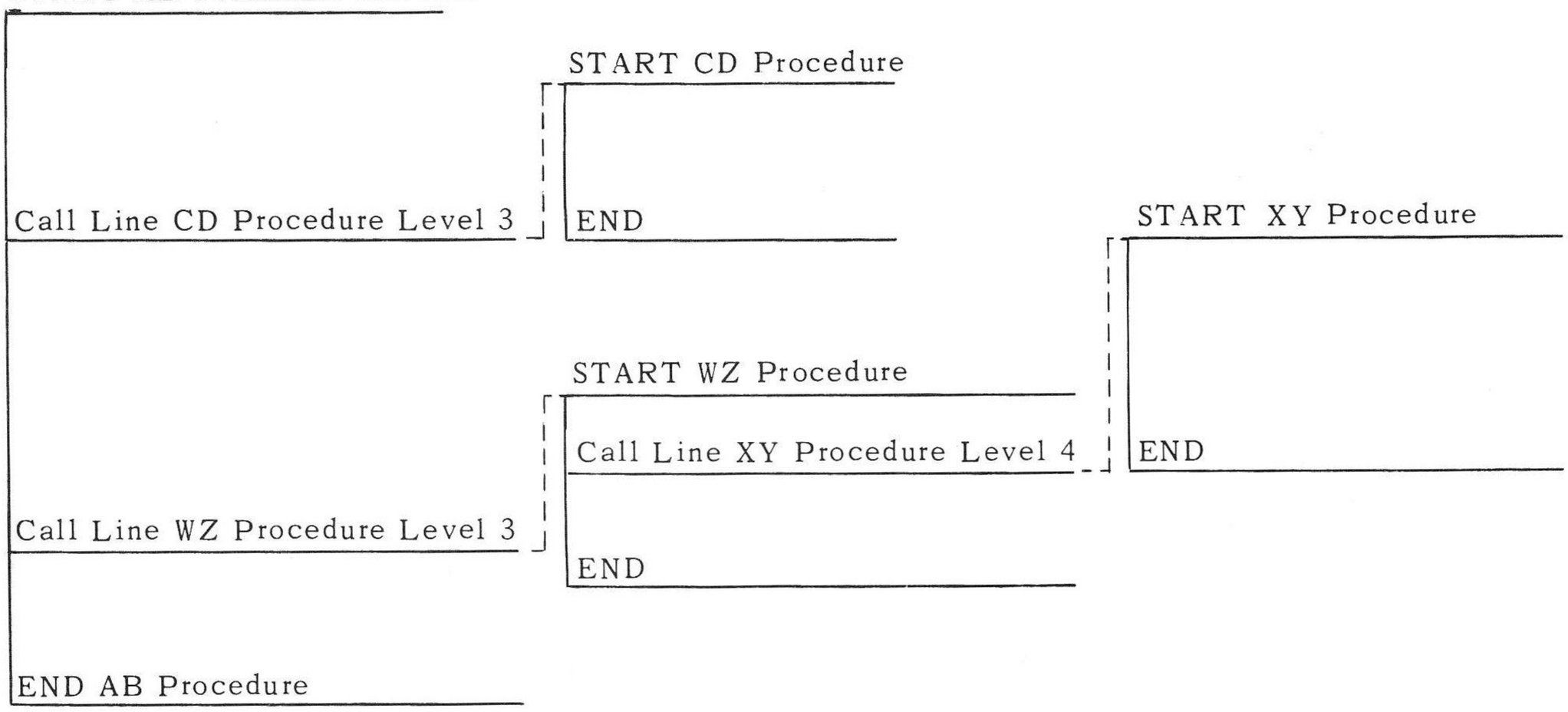
Explanation:

Procedures CD and WZ are nested within the XY procedure and the XY procedure is nested within the AB procedure.

Procedure XY cannot be referenced unless procedure AB is referenced first. Procedures CD and WZ may not be referenced unless procedure XY is referenced first.

3.1.7.2. Implied or Logical Nesting

A procedure which is called upon within another is an implied procedure if the procedure referenced is not nested physically with another procedure. Only the call line is located within the higher procedure. Implied nesting may be nested a maximum of 62 levels.



The PROC sample must appear physically before its reference in the main program.

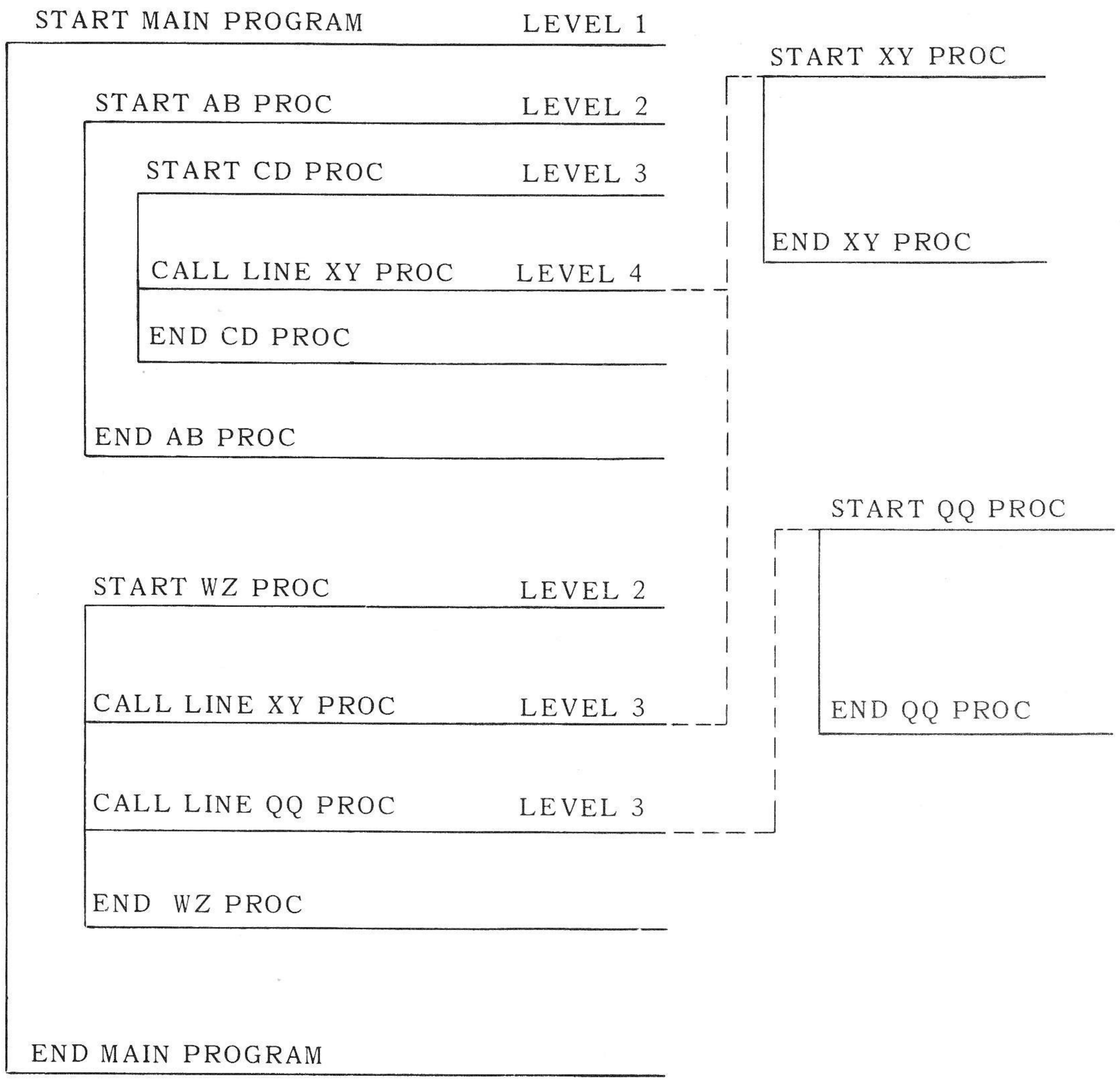
If a GO statement transfers control to an entrance label of another procedure, this is not considered nesting but is a lateral transfer and does not change levels (see 3.2.3).

3.1.7.3. Levels of Procedures

When procedures are nested, they are considered to have various levels of hierarchy. The main program is considered Level 1. A procedure nested physically within the main program is Level 2. A procedure physically nested within a Level 2 procedure is considered to be one level lower, Level 3. Procedures may be nested down 62 levels.

When a procedure is logically nested within another procedure, it is considered one level lower than the procedure in which the call line appears.

Example:



3.1.8. Procedure Labels

A label is a symbolic representation of some value. A label may be either local or external. An external label is one which may be referenced by other programs or procedures. A local label is one whose value is restricted to the program or to the procedure in which it appears.

Labels appearing in the main program may be referenced by any procedure. Labels appearing in a procedure normally cannot be referenced by the main program or other procedures. Nesting of procedures creates a hierarchy of labels. Procedure labels of one level may not be referenced by a procedure of another level.

An asterisk immediately following a label makes that label available for reference from a procedure one level higher. A label at the main program level having an asterisk is externalized and may be referenced by any other program. A label in a nested procedure may be referenced by one procedure level higher for each asterisk appended to the label.

Below is a diagramatic representation of labels appearing in procedures.

Example:

```
START MAIN PROGRAM

CALL 1  MAJOR

        MAJOR* PROC

                CALL 2   MID

                        MID*  PROC

                        CALL 3   MINOR

                                MINOR * PROC

                                M1
                                M1A**
                                M1B***

                                END

                        M2
                        M2A*

                        END

                M3
                M3A*

                END

END MAIN PROGRAM
```

Explanation:

As a rule, lower level labels are available only after a call is made on the PROC in which that label appears.

- The main program may reference PROC MAJOR, but not MID and MINOR.
- Label M3 is available to the MINOR, MID, and MAJOR PROC's but not to the main program.
- Label M3A is available to all the PROC's and the main program.
- Label M2 is available to the PROC's MINOR and MID.
- Label M2A* is available to the MINOR, MID, and MAJOR PROC's.
- Label M1 is available only to the MINOR PROC.
- Label M1A** is available to the MINOR, MID, and MAJOR PROC's.
- Label M1B*** is available to all PROC's and the main program.

Because they are physically nested, the main program cannot reference the MID and MINOR PROC's.

3.1.8.1. Externalizing Procedure Labels

A label defined within a program may only be referenced within that program. To make a procedure label available to other programs, it must be equated to an external label. Externalizing a procedure label is done in the main program. For example

```
M3*     EQU     M3A
```

equates procedure label M3A in the main program to the externalized label M3* and makes M3A available to other programs. The label M3A must be known to the main program level.

3.1.9. Forward References

Forward references occur when a label is referenced prior to being encountered by the assembler. Forward references also occur when a label has been referenced whose value is dependent upon values not yet encountered by the assembler. Forward references are prohibited when using assembler directives.

```
1.      RES     A
2. A    LA      1,2,3
```

The operand A of the RES directive is a forward reference. The value of A will never be evaluated and is considered undefined.

The user is cautioned against basing the generation of code within a procedure sample on a condition involving a forward reference. Consider a hypothetical MOVE procedure. The programmer may check if the move from and move to addresses are the same. On the first pass through the source data, the labels of the from and to areas may or may not have been defined. On the second pass of the assembler, the labels will have been defined. The values reached on each pass of the assembler can be different.

If the procedure sample does choose an error exit on pass one (that is, no generation of code) and does produce code on pass two, the labels following the call on the sample are assigned a location counter value on pass one that is different in pass two. The result is a multiple definition of those labels.

Whenever the assembler gets a different line count on the first or second pass, multiple definitions of succeeding labels occur and D error flag is set.

The user is admonished to take great care when using forward references.

3.1.10. Control Counter on a Procedure

A procedure may be made to generate code under a particular location counter by specifying the counter on the PROC directive line in this manner:

```
LABEL          OPERATION
$(3),P         PROC   1,3
```

The assembler considers this a local control counter declaration. Lines following the procedure reference are under the control counter used prior to the procedure reference.

Example:

```
1.  $(2),ADDP*       PROC 1
       .
       .
       .
7.     END
8.  $(1),ABC    LA   A9,   TAG
       .
       .
20. CALL2       ADDP*
       .
       .
```

Explanation:

Code produced between lines 1 through 7 is under control counter 2.

Code at lines 8 through 19 is under control counter 1.

Code produced at line 20 by procedure ADDP is under control counter 2.

3.1.11. Hierarchy of Label Definition

As the assembler encounters each label, it tries to evaluate each label as best it can. It then stores the label and its value in an appropriate table. Since there are several different tables serving several different functions, the assembler looks up the definition for any label in a definite predefined order. This can be called a hierarchy of reference.

While it is possible to define a label in many ways, it is the hierarchy of reference that brings back one or the other definition.

Even within the same type of structure, hierarchial look-up is employed. For example, a procedure called SQRT may exist within the user's library or within the system library. A call for SQRT causes a reference to be made to (1) the program library, (2) the user's library, or (3) the system library, in that order.

3.1.12. Waiting Labels

A label may be affixed to the line of reference to a procedure. Under normal conditions, this label is defined as equal to the value of the current location counter at the time of the procedure call. It is possible to associate this label with a line within the procedure. This is done by coding an asterisk (*) alone in the label field of that particular line in the procedure. The label of the calling line is processed exactly as though it had appeared in place of the asterisk except that it is defined at the level of the reference line on which it appeared.

Example:

```
X*      PROC    1,2
        TLEM    X(1,1),4,,11
*       J       $+3
        END
RAM     X       14
        LA      17,RAM
```

In this example, RAM is the address of the J line. If this line had not included the asterisk in the label field, RAM would have been the address of the TLEM line.

3.1.13. Permanency of Label Definition

Essentially the duration of a label definition is dependent on:

- The frequency with which a label is redefined.
- The length of time the label exists.

The frequency of redefinition is a function of the application:

```
1   LABEL     Δ     OPERATION     Δ     OPERAND
COUNT       DO     10,+    COUNT
```

COUNT is redefined for each line of code.

```
COUNT       DO     10,     CALL ALPHA BETA GAMMA
```

The duration of any definition of the label COUNT is dependent on the CALL and any subcalls CALL may include.

The existence of a label is dependent on both its physical and logical localities.

If a label is confined to a procedure, it is available only during the subassembly of that procedure. While a label may be physically confined to any assembly or subassembly, an appended asterisk may flag it to be available one level higher. This makes the definition more durable or permanent. Therefore, labels confined to a procedure may be defined differently each time the procedure is called, but should not be redefined if their logical locality implies permanence. For example:

```
CALL*       PROC
VALUE       +       CALL(1,1)
            END
```

Multiple calls on the above procedure would result in an error since VALUE is ambiguously redefined.

3.1.14. Noise Words

Assume that three calls on a procedure are made in succession.

```
1    LABEL     Λ     OPERATION     Λ     OPERAND
CASE1          ADD   25 TO MAJOR
CASE2          ADDE  MAX TO MAJOR+3
CASE3          ADDE  13 TO MINOR,10
```

The word TO is a noise word which is included to improve the readability of the line. It must be equated to 0 prior to using it in the call. Each such noise word is counted as a parameter; in these examples, it is the unused field CT(2,1).

3.2. COMPLEX PROCEDURES

3.2.1. DO Directive, DO

The DO directive in the assembler is a powerful tool which, when used within procedures, provides great flexibility and power.

The format of a DO line is:

LABEL1 DO EXPRESSION , LABEL2 OPERATION OPERAND

The comma divides the DO line into two parts:

(1) the determinant: LABEL1 DO EXPRESSION

(2) the DO-item: LABEL2 OPERATION OPERAND

The expression following the DO directive determines how many times the DO-item is generated. LABEL1 is optional; if used, LABEL1 serves as a counter reference reflecting the current number of times the DO-item has been executed.

The DO-item may be any symbolic line of coding. The DO-item may contain another DO directive, nested down to eight levels.

Example of a simple DO:

```
NUMBER ,
A      DO   5 , ‡A
```

As each +A is generated, the value of the label A is increased by +1 until 5 DO-items are generated.

All expressions appearing in the determinant must be defined. All undefined expressions are 0 until they are defined. If, for example, the determinant A DO TAG is written and TAG has not been defined, the assembler reads A DO 0. A line count is not assigned and the DO is not executed. The DO count must be defined before the DO directive is encountered.

3.2.1.1. Conditional DO

The operators <=> are relational operators and generate the values 0 or 1. If the relationship between two expressions is true, the value of the expression is 1. If false, the value is 0.

The expressions employed in the determinant have, to this point, been simple. A more complex expression would be a conditional DO:

```
1    LABEL     Δ     OPERATION          Δ          OPERAND
Q              DO    A=1,ADD 4 TO MINOR
```

If A is 1, then by substitution 1=1. Since the relation is true, the value 1 is substituted for the expression. The result is equivalent to:

```
Q              DO    1,ADD 4 TO MINOR
```

The DO-item would be executed once.

The following move procedure illustrates the conditional DO. The procedure will move data from one area to another or initialize any given area with any given constant.

Call line for MV PROC has five fields. The fifth field has two subfields.

```
    MOVE,12 WORDS TO WRITE FROM INPUT,2
```

	LABEL	OPERATION	OPERAND
1.	MV	PROC	
2.	MOVE*	NAME	1
3.	ZERO*	NAME	0
4.		LA	A0,(1,MV(3,1))
5.		DO	MV(3,2)>0 , AA 12,0,MV(3,2),14
6.		LA	A1,(MV(0,0),MV(5,1))
7.		DO	MV(5,2)>0 , AA 13,0,MV(5,2),14
8.		LR	0101,MV(0,1),,14
9.		BT	A0,0,*13
10.		END	

The DO in lines 5 and 7 check to see if a parameter was supplied. In this case, the parameter specifies an index: the procedure is checking to see if the address was indexed. If it was, it would add the value of the index to the address to achieve the proper effective address. The paraform MV(0,1) references the subfield attached to the operation.

Note that unspecified subfields of a given field are automatically assigned a value of zero. In this example, the third field,WRITE, is such a case. Therefore, in line 5 of the coding MV(3,2)>0 is false because lack of the subfield for WRITE reduces the value to 0, and 0 is not greater than 0.

LABELA may not be used as the expression value in the determinant.

```
E  DO  (ARRAY (1,E)>0 , AA  A12, MIN
```

This example cannot be executed. The value for E is initially 0 and the item ARRAY1,0 does not exist. If written as follows, however,

```
E  DO 1, DO (ARRAY (1,E)>0, AA A12, MIN
```

the line of coding can be executed since ARRAY (1,1) is referenced.

3.2.2. The NAME Directive, NAME

The NAME directive has three functions:

(1) It provides a local reference point within a given procedure or function.

(2) It acts as an alternate entrance into the procedure or function.

(3) It may give a value to the procedure. It must be located between the PROC or FUNC line and its associated END line.

When assigning a value, the value is written as the operand of the NAME line and becomes meaningful as the *0th* subfield, *0th* field if and only if the procedure is entered at this name line. If such a value exists, this counts as an additional field to the procedure. A paraform may be used to represent this value if the NAME line is within a nested procedure. Additional subfields may be added to this *0th* field at the time of the call.

Example:

```
    LABEL   Δ  OPERATION  Δ           OPERAND                       Δ   COM
 1. SEE*       PROC       4
 2. SAW*       NAME       2
 3.            LA         SEE(1,1),SEE(1,2),SEE(0,0),SEE(0,1)
 4.            TLE        SEE(2,1),SEE(3,1)
 5.            GO         EYE
 6. EAR*       NAME       4
 7.            SA         SEE(1,1),SEE(0,0),SEE(0,1)
 8. EYE*       NAME
 9.            DO         SEE(4,*1) , +3
10.            END
11.
12.
13. CALL1      SEE        16,CAT 17 DOG *43
14. CALL2      SAW,5      16,CAT 17 DOG 43
15. CALL3      EAR,6,7    16
```

Explanation:

Line 13 calls the SEE PROC. The subfields in the operand field are SEE (1,1), SEE (1,2), SEE (2,1), SEE (3,1), and SEE (4,1). The subfields refer to 16, CAT, 17, DOG, * 43, respectively. These values are substituted for corresponding paraforms in the procedure and lines 3 and 4 become:

```
LA     16,   CAT,  0,  0

TLE    17,   DOG
```

SEE (0,0) and SEE (0,1) equal 0 since the entrance was not made at the NAME line. Lines 6 and 7 are not processed.

Line 14 calls the SAW PROC. The values SEE (0,1), SEE (1,1), SEE (1,2), SEE (2,1), SEE (3,1) and SEE (4,1) refer to 5, 16, CAT, 17, DOG and 43, respectively. The paraform SEE (0,1) is 5 as 5 is the second subfield of the 0*th* field. SEE (0,0) references the NAME line and its value is 2. Lines 3 and 4 become:

```
LA    16,    CAT, 2, 5

LA    17,    DOG
```

Lines 6 and 7 are not processed.

Line 15 calls the EAR PROC. This causes entrance at the NAME line labeled EAR. SEE (0,0) is 4, SEE (0,1) is 6, SEE (0,2) is 7, and SEE (1,1) is 16. Line 7 becomes:

```
SA    16,    4,    6
```

The DO statement generates no coding.

3.2.3. The GO Directive, GO

This directive transfers control of the assembler to the line whose label is in its operand field. This label must be one of the following:

- A label of a NAME directive in the same procedure. If the transfer is a forward or downward one, the label must be asterisked.
- An external label of a NAME directive of any procedure.

Example:

```
1.  DONC          PROC
2.  D1*           NAME  0
3.  D2*           NAME  1
4.                DO    DONC(0,0)=0,    GO OUT
5.                SZ    CAT
6.  OUT*          NAME
7.                END
```

Explanation:

If the program is entered at D1 NAME, the value of DONC (0,0) is 0. The conditional DO directive on line 4 is true and control transfers to OUT.

If the program is entered at D2 NAME, the value of DONC (0,0) is 1 and the conditional DO directive on line 4 is executed (conditional DO directive is false). The SZ CAT instruction on line 5 is executed.

The GO directs the assembler to different sections of sample code to be assembled. The sections of code referenced need not be in the procedure currently being executed. The subassembly process is avoided, thereby substantially reducing assembly time.

Example:

```
1 LABEL    Δ OPERATION    Δ OPERAND
1.  AB*        PROC
2.             DO     X=1,GO CONT1.OPTION1
3.             DO     X=2,GO CONT2.OPTION2
4.             ****                .OPTION3
5.             ****
6.             END
7.  AB1*       PROC
8.             ****
9.             ****
10. CONT1*     NAME   . THIS CODE CAN BE
11.            ****   . REFERENCED BY THE
12.            ****   . CURRENT PROCEDURE
13.            END    . AND AB PROCEDURE
14. AB2*       PROC
15.            ****
16.            ****
17. CONT2*     NAME   . THIS CODE CAN BE
18.            ****   . REFERENCED BY THE
19.            ****   . CURRENT PROCEDURE
20.            END    . AND AB PROCEDURE
```

NOTE: The four asterisks appearing alone on a line (****) represent lines of miscellaneous coding.

A call on procedure AB can optionally develop code from its own sample or that of AB1 or AB2. The GO does not cause subassembly. If the AB were rewritten as

```
1   LABEL     Δ     OPERATION     Δ     OPERAND
AB*           PROC
              ****
              ****
              DO        X=1,    CONT1
              DO        X=2,    CONT2
              DO        X<3,    END     RETURN
              ****
              ****
              END
```

then CONT1 or CONT2 would cause subassembly. The effect is the same, but the time for assembly is longer. The return line is included because termination of subassembly is desired at the return point. The return DO-line assumes X is never 0 or negative.

3.2.4. Procedure Modes

Procedures can be developed in any one of three modes: simple, generative, or interpretative.

3.2.4.1. Simple Mode

The simple mode occurs when the object procedure developed is equivalent to the object procedure declared.

Simple Mode Procedure:

```
CTR*          PROC
              LA      12,CTR(1,1)
              AA      12,CTR(1,2)
              SA      12,CTR(1,1)
              END
              CTR     TALLY,  13
```

The above PROC declares and generates three lines of code which add a given value to a given counter.

3.2.4.2. Generative Mode

The generative mode occurs when the object procedure developed is a multiple of the object procedure declared.

Generative Mode Procedure:

```
1    LABEL     Δ     OPERATION     Δ     OPERAND     Δ
CTR*           PROC
SUBCT*         PROC
               LA      12,SUBCT(1,1)
               AA      12,SUBCT(1,2)
               SA      12,SUBCT(1,1)
               END
Q              DO      CTR,  SUBCT CTR(Q,1),CTR(Q,2)
               END
CALL1          CTR     TALLY,13 TOLL,5 TOTAL,2
```

The call line supplies three fields; three lines of code are generated for each.

3.2.4.3. Interpretative Mode

The interpretative mode occurs when the object procedure declared interprets the fields given and generates code based on the interpretation.

Interpretative Mode Procedure:

```
P              PROC
SA*     NAME   01
LA*     NAME   010
SNA*    NAME   02
I$      FORM   6,4,4,4,2,16.   1108  INSTR FORMAT
        I$     P(0,0)P(0,1)+P(1,4),P(1,1)-12,;
               P(1,3),2*P(1,*3)+P(1,*2),P(1,2)
        END
```

The six fields in the I$ form reference line represent the f, j, a, x, h-i, and m fields, respectively. It can be seen that, by interpretation, one procedure may define a group of instructions of the same general type.

3.3. SPECIAL APPLICATIONS

3.3.1. Instruction Word Generation

An example of how a normal instruction word may be generated using a procedure is shown here for the LA instruction.

```
LOAD    PROC    1,1
LA*     NAME    010
F       FORM    6,4,4,4,2,16
        F       LOAD(0,0),LOAD(0,1)+LOAD(1,4),;
                LOAD(1,1)-12,LOAD(1,3),;
                2*LOAD(1,*3)+LOAD(1,*2),LOAD(1,2)
        END
```

- One field is expected and one word will be generated.
- LOAD (0,0) is the value on the NAME line which is the function code of the LA instruction.
- LOAD (0,1) and LOAD (1,4) are the two arbitrary positions in which the j designator may be coded.
- LOAD (1,1) is the expected a designation.
- LOAD (1,3) is the index designation. LOAD (1,*3) will set a 0 or 1 bit in the incrementation designator field (h field).
- LOAD (1,2) is the u portion of the instruction. LOAD (1,*2) will set a 0 or 1 bit in the indirect addressing field (i field).

3.3.2. ARRAY Generation

The following test describes an assembler procedure which generates an array having one, two, or three dimensions. Each element in the array may be one or more words. Reference to the procedure, named ARRAY, generates an array with the label and dimensions specified, in a manner analogous to the DIMENSION statement in FORTRAN.

Following generation of an array in the assembler code, reference may be made to elements of the array by suffixing the array name with appropriate subscripts enclosed in parentheses.

Arrays may be generated with the following procedure call line format:

label ARRAY columns,rows,pages words-per-element

where label is the name to be given to the array. The first field contains three subfields: columns, rows, and pages which are expressions with integer values for each dimension. Words-per-element is an integer value expression specifying the number of words for each element in the array. If this parameter is omitted, words-per-element is assumed to be 1. The number of parameters on the first parameter list determines the number of dimensions.

A sample ARRAY procedure follows:

```
    LABEL    Δ  OPERATION    Δ  OPERAND          Δ  COMMENTS
1.  P           PROC 2
2.  ARRAY*      NAME 0
3.  K(1)        EQU  1          .INITIAL DEFINITION PAGE COUNTER
4.  Z           NAME 0
5.  J(1)        EQU  1          .INITIAL DEFINITION ROW COUNTER
6.  Y           NAME 0
7.  I(1)        EQU  1          .INITIAL DEFINITION COLUMN COUNTER
8.  X           NAME 0
9.              DO   P(1)=1,L(I(1))      RES 1
10.             DO   P(1)=2,L(I(1),J(1))  RES 1
11.             DO   P(1)=3,L(I(1),J(1),K(1))  RES 1
12.             DO   P(2,1)>0,  RES P(2,1)-1
13. I(1)        EQU  I(1)+1          .bINCR COLUMN COUNTER
14.             DO   P(1,1)+1>I(1),  GO X  .bMORE COLUMNS
15. J(1)        EQU  J(1)+1     .ROW FINISHED, INCR ROW COUNTER
16.             DO   P(1,2)+1>J(1),  GO Y  .bMORE ROWS
17. K(1)        EQU  K(1)+1     .bPAGE FINISHED, INCR ROW COUNTER
18.             DO   P(1,3)+1>K(1),  GO Z  .bMORE PAGES
19. *           EQU  L
20.             END
```

Explanation:

The asterisk in column 1 of line 19 is a waiting label. Line 19 is assigned the label of the call line.

In most cases, the RES directive should be used as in lines 9, 10, and 11. Another way of coding the three generating lines is presented below.

```
9.              DO   P(1)=1,L(I(1)),+I(1)
10.             DO   P(1)=2,L(I(1),J(1)),+I(1)*J(1)
11.             DO   P(1)=3,L(I(1),J(1),K(1)),+I(1)*J(1)*K(1)
```

A reference to this procedure would appear as:

```
W               ARRAY 3,4
```

A 3 by 4 array would be generated with element storage by row, for example, consecutive cells of main storage would contain the three columnar positions of row 1 followed by the three columnar positions of row 2, and so on, through row 4.

For purposes of demonstration, the procedure may be coded to insert the product of the row and column indexes of each element in each element. The preceding line would produce the following array, assuming generation under a location counter with the initial value 0100:

ADDRESS	VALUE	ADDRESS	VALUE
000100	1	000106	3
000101	2	000107	6
000102	3	000110	9
000103	2	000111	4
000104	4	000112	8
000105	6	000113	12

If visualized as a matrix, the logical element arrangement would be as follows:

	COLUMNS		
ROWS	1	2	3
	2	4	6
	3	6	9
	4	8	12

The effect of the procedure is to equate a series of values of a location counter with a series of subscripted labels. Using such a label in an assembler instruction or expression causes the subscripted label to be replaced by the location of the array element it specifies.

Based on the array shown above, here are examples of references to a generated array:

LABEL	VALUE (its address)
A (1,2)	000103
A (2,3)	000107
A (3,4)	000113

An actual usage might be:

```
          1         2         3         4         5         6         7
0123456789012345678901234567890123456789012345678901234567890123456789
          LA      12,A(3,2)
```

which would be the same as writing:

```
          1         2         3         4         5         6         7
0123456789012345678901234567890123456789012345678901234567890123456789
          LA      12,105
```

A one-dimensional array of four two-word elements could be generated with the line:

```
          1         2         3         4         5         6         7
0123456789012345678901234567890123456789012345678901234567890123456789
B               ARRAY   4   2
```

Assuming generation under a location counter with the initial value 0200 and the vector expressed in each element, the following could be produced:

ADDRESS	VALUE
000200	1
000202	2
000204	3
000206	4

Locations 0201, 0203, 0205, and 0207 are the second words of each element. It is important to note that for an array with multiword elements, a reference to one of the subscripted labels produces the address of the leftmost (most significant) word of the element. Here are examples of references to this array:

LABEL	VALUE (its address)
B(1)	000200
B(3)	000204
B(3) + 1	000205

3.3.3. Display Console Linkage

The procedure presented below provides linkage to the display console routine in the 1106/1108 Operating System. It will display one word and can be used to indicate errors.

```
1    LABEL      Δ      OPERATION      Δ      OPERAND
1.   TYPE              PROC
2.   TYPE*             NAME
3.                     LMJ     11,KTYPE$
4.   K                 FORM    6,12,18
5.                     K       0,6,$+2
6.                     J       $+2
7.                     +       TYPE(1,1)
8.                     END
```

- Lines 1 and 2 illustrate that labels and entry points do not conflict.
- Line 4 contains a FORM directive allowing the programmer to specify the number of bits to be contained in each field whenever the form is referenced.

For example, the call:

```
OFLO           TYPE    +('OFLO')
```

would generate the expanded code:

```
1.  OFLO           LMJ     11,KTYPE$
2.  K              FORM    6,12,18
3.                 K       0,6,$+2
4.                 J       $+2
5.                 +       OFLO'      . THE PLUS SIGN PADS
6.                                    ZEROS LEFT OF OFLO
```

Line 3 indicates: display 6 characters from line 5.

The relocatable output is (assume location counter is at 000042 when the call is made):

000042	74	13	13	00	0	0	001032	Instruction
000043	00	0006			000045			K format (6,12,18)
000044	74	04	00	00	0	0	000046	Instruction
000045	000024132124							= Fieldata b b OFLO

The routine can be used to provide a form of trace when the logic of a program is in question. It is a simple matter to install or remove a number of the following calls.

```
1  LABEL     Λ   OPERATION     Λ   OPERAND
TYPE          +('POINT1')
TYPE          +('POINT2')
TYPE          +('POINT3')
TYPE          +('POINT4')
```

3.3.4. Print Linkage PROC

As an alternative to using the display console, the system provides the print linkage PROC for using the printer. It is used as follows:

```
          P$RINT   ('OFLO'),1    . THE SECOND PARAMETER CAUSES
          P$RINT   ('POINT'),1   . PRINTING OF ONE WORD
```

Sometimes a blank line is required; the following PROC provides it.

```
LINES     EQU   0
LINE      EQU   1
BLANK*    PROC
          P$RINT   ('bbbbbb'),1,BLANK(1,1)  . bBLANK(1,1) INDICATES THE
          END                               . bNUMBER OF LINES TO BE SKIPPED
```

The call

```
CASE1     BLANK     LINE
```

causes the printer to skip one line and the following call

```
CASE 2      BLANK    2     LINES
```

causes the printer to skip two lines. The words LINE and LINES are noise words used to improve readability.

The labels LINES and LINE are equated to values to prevent undefined flags from occurring and to provide the proper expressions in line 4. In CASE1, BLANK (1,1) that is, LINE provides a value of 1 and in CASE2, BLANK (1,1) references 2. One line of blanks is printed. In CASE2, the constant 2 causes an extra line to be skipped after printing the blank line.

3.3.5. Example of a Procedure Listing

The following listing from an 1106/1108 assembly includes examples of procedure structure, nested procedures, and procedure references. The coding produced by reference to M PROC determines the largest or smallest value in a series of values. Each value is assumed to be represented in a 36-bit signed word. Following the listing is an explanation of the action taken by the assembler while processing this coding.

```
000001  000000                          RES 01000-$
000002                          M       PROC
000003                          MAX*    NAME 0
000004                          MIN*    NAME 1
000005                          M1*     PROC 0
000006                                  DO   M(0,0)=0 , TLE M(1,1),M(I+2,1),M(I+2,2)
000007                                  DO   M(0,0)=1 , TG  M(1,1),M(I+2,1),M(I+2,2)
000008                                  LA   M(1,1),M(I+2,1),M(I+2,2)
000009                                  END
000010                                  LA   M(1,1),M(2,1),M(2,2)
000011                          I       DO   M-35, M1
000012                                  END
000013          000000010000    L       EQU  010000
000014  001000 10 00 04 01 0 010000     MAX  16 L,1 L+2,1 (12)
        001001 54 00 04 01 0 010002
        001002 10 00 04 01 0 010002
        001003 54 00 04 00 0 001012
        001004 10 00 04 00 0 001012
000015  001005 10 00 04 01 0 010000     MIN  16 L,1 L+2,1 (12)
        001006 55 00 04 01 0 010002
        001007 10 00 04 01 0 010002
        001010 55 00 04 00 0 001012
        001011 10 00 04 00 0 001012
000016          000000000000            END
        001012 000000000014
```

Line 1 sets the controlling location counter to 1000_8.

Lines 2 thru 12 the body of the procedures; these lines are temporarily stored by the assembler for later reference.

Line 13 equates L to the value $10,000_8$.

Line 14 is a reference line to PROC M. It contains four fields. List 1 has one subfield; fields 2 and 3 each have two subfields; field 4 has one subfield, the literal 12. Coding produced by the reference to the procedure is shown to the left of the reference (addresses 001000-001004).

Line 2, the first line of M PROC, is referred to through MAX NAME 0, line 3.

Line 10, the first line of M PROC to produce coding, creates the first instruction at address 001000. The operand entries of this instruction are determined by subfields supplied by the reference on line 14.

Line 11 references the nested procedure M1; the number of references to M1 PROC is determined by the expression M–3.

Line 5, the first line of M1 PROC, has a 0 in the operand field indicating that no list is to be submitted to M1 when it is referenced.

Line 6 produces a TLE instruction (54) at address 001001 since MAX was the entry to PROC M. The counter I of the DO line (line 11) within M PROC advances the field number and thus accesses the appropriate subfield for use in the compare instructions.

Line 7 is skipped on this iteration because the condition M(0,0)=1 was not met.

Line 8 produces a LA (10) instruction at address 001002 in the same manner as line 10.

Line 9 terminates this iteration of M1 PROC.

Line 11 now references M1 PROC for the second iteration. Lines 5 through 9 are executed as above.

Line 12 terminates M PROC. Assembly continues at line 15.

Line 15 is another reference to M PROC. The execution is identical except that line 6 is skipped and line 7 is executed.

Line 16 terminates the assembly or program.

3.4. THE FUNCTION DIRECTIVE, FUNC

The function directive is a device within the assembler which saves certain predetermined lines of coding as they are encountered during assembly. When referenced subsequently during the assembly, a quantity computed according to this coding is substituted for the reference call within the program.

The FUNC is similar to the PROC in that the lines of coding representing the definition must precede any call (reference point) and this delineation of code is saved when encountered. The two differ in that a value is calculated when a function is referenced and no object lines of coding are ever generated. The function operates entirely at assembly time and stores its results into the program at this time.

The general rules of definition are similar to those for the PROC directive. A FUNC directive must start the definition area and the line must be labeled. If the line is to be an entry point into the function, the label must have an asterisk. The delineation of code is terminated with an unlabeled END directive which may have an operand. This operand field is an expression whose evaluation results in substitution of the proper quantity into the reference point in the program. The FUNC may produce either single or double precision values depending on the type of operations employed within the FUNC.

NAME lines with starred labels may be used as alternate entry points into the FUNC. NAME lines may also be used as local reference points within the FUNC. Forward references should be avoided.

The coordinate system of input is a single field of n subfields. The reference point is of the form LABEL (a,b,...,n) where LABEL is the FUNC or NAME line label and a,b,...,n are input values. This reference point can be found imbedded within an expression or can be the entire expression itself.

LABEL (0) is meaningful as a paraform if entry to the function is made through a NAME line. This input value is the operand of the NAME line.

A particular subfield is referenced within a FUNC by writing the FUNC label followed by one expression enclosed in parentheses. This expression specifies the ordinal number of the subfield within the field.

Either PROC's or FUNC's may be nested within a FUNC provided the procedure is not a line generating one. They are usually nested so that labels can be redefined at different levels. All the rules of nesting discussed in paragraph 3.1.7 apply to FUNC.

A typical application of the FUNC directive is the case where a certain average calculation is made throughout the coding. This calculation could be made manually and is not dependent upon the execution of the object code. Let a be the number of the first type of object and b its unit price; c is the number of the second type object and d is its unit price. A mathematical expression to calculate the average price of the combined number of objects would be:

$$\text{Average cost} = \frac{ab + cd}{a + c}$$

Assume that values of a, b, c, and d are known at assembly time to be 5, 10, 8, and 12, respectively. The calculation is as follows:

```
AVGCOS*  FUNC
A(1)     EQU   AVGCOS(1)*AVGCOS(2)
B(1)     EQU   AVGCOS(3)*AVGCOS(4)
C(1)     EQU   A(1)+B(1)
D(1)     EQU   AVGCOS(1)+AVGCOS(3)
         END   C(1)/D(1)
```

Although the entire expression could be calculated in one step, it is faster and more expedient to break up the expression into subexpressions and then to combine them.

```
         LA    12,AVGCOS(5,10,8,12),,016
```

This line contains the reference which will cause generation of the value at assembly time.

The following FUNC source code statement includes examples of a FUNC structure, a nested procedure, and function references. The value produced by reference to SQRT FUNC is the square root of the largest square which is less than or equal to the parameter provided in the reference. Following the example is an explanation of the action taken by the assembler while processing this coding.

```
000001                          SQRT*   FUNC
000002                          A(1)    EQU   0
000003                          B(1)    EQU   0
000004                          C*      PROC  0
000005                          A*(1)   EQU   B(1)*B(1)
000006                          B*(1)   EQU   B(1)+1
000007                                  END
000008                          D       NAME
000009                                  C
000010                                  DO    SQRT(1)>A(1),GO D
000011                                  END   B(1)-(SQRT(1)<A(1))-1
000012  00  000000  000000000010        + SQRT (64)
000013      000001  000000000006        + 2*SQRT (13)
000014              000000000000          END
```

Lines 1 thru 11 the function with a nested procedure is temporarily stored by the assembler for later reference.

Line 12 if a reference to SQRT FUNC, introduced above. The reference provides one subfield (64). The object line produced by the reference would contain an octal value 000 000 000 010.

Line 1 is the entrance to the FUNC.

Line 2 equates a value of 0 to the subscripted label A(1).

Line 3 equates a value of 0 to the subscripted label B(1).

Line 9 is a reference to C PROC.

Line 4 is the entrance of C PROC. The 0 operand expression indicates that no field is to be submitted to C PROC when referenced.

Line 5 equates a value to the label A(1). The value produced is a result of the operand expression, and will be an ascending sequence of squares $(0,1,4,9,\ldots,e_n)$.

Line 6 equates a value to the label B(1). The value produced is a result of the operand expression, and will be an ascending sequence of square roots $(0,1,2,3,\ldots,e_n)$.

Line 7 terminates this iteration of C PROC.

Line 10 compares the value of the SQRT subfield (64) to the value of A(1). If it is greater, the GO line will be executed once. Assembly continues at line 8.

Line 8 is a NAME entry point.

Line 9 references C PROC for the second iteration.

If the SQRT parameter value is not greater than the value of A, assembly continues at line 11.

Line 11 terminates SQRT FUNC. The operand expression provides the value of SQRT FUNC for this reference.

Line 13 is another reference to SQRT FUNC. The execution is identical. The object line produced by this reference would contain an octal value 000 000 000 006.

Line 14 terminates the assembly or program.

APPENDIX A. ABBREVIATIONS AND SYMBOLS

CONVENTIONS

Abbreviations and symbols frequently used in the description of the instruction repertoire are given below:

( )	Contents of register or address within parentheses.
( )	Complement of contents of register or address.
|( )|	Absolute value or magnitude.
$( )_{17\text{-}00}$	Subscripts indicate the bit positions involved. A full word is normally not subscripted. Subscripts are also used to designate octal or decimal notation.
( )c	Floating point biased exponent.
( )f	Final contents.
( )i	Initial contents.
( )m	Floating point fixed point part.
( )j	j-designated portion.
f	Function code.
j	Partial word designator or function code extension.
a	Arithmetic register designator. In input/output instructions, "a" designates an I/O channel.
A	Arithmetic Register.
x	Index register designator.
x_a	Index register designator in a-field.
X	Index Register.
X_a	Index Register specified by coding x_a.

X_M	Modifier portion of an index register.
X_I	Increment portion of an index register.
r	Same as r_a.
r_a	Designator specifying an R Register. It is coded in the a-designator position of an instruction word.
R	R Register.
R_a	R Register specified by coding r_a.
u	The base address of the operand (or the actual operand) as coded in u-field of an instruction.
U	The effective address or value of the operand after application of indexing and indirect addressing.
U_d	Destination address.
U_s	Source address.
h	h-designator of the instruction word. A value of 1 specifies incrementation of an index register.
i	i-designator of the instruction word. A value of 1 specifies indirect addressing.
PSR	Processor State Register.
AND	Logical product, or logical AND.
OR	Logical sum, or inclusive OR.
XOR	Logical difference, or exclusive OR.
→	Direction of data flow.

APPENDIX B. INSTRUCTION REPERTOIRE

Table B-1 lists the 1106/1108 instruction repertoire in function code order. Table B-2 cross-references the mnemonic and function code.

Function Code (Octal) f	Function Code (Octal) j	Mnemonic	Instruction	Description ②	1108 Execution Time in μsecs.①	1106 Execution Time in μsecs.⑥
00	–	–	Illegal Code	Causes illegal instruction interrupt to address 241_8	–	–
01	0-15	S, SA	Store A	(A)→U	.75	1.5
02	0-15	SN, SNA	Store Negative A	−(A)→U	.75	1.5
03	0-15	SM, SMA	Store Magnitude A	|(A)|→U	.75	1.5
04	0-15	S, SR	Store R	(R_a)→U	.75	1.5
05	0-15	SZ	Store Zero	ZEROS→U	.75	1.5
06	0-15	S, SX	Store X	(X_a)→U	.75	1.5
07	–	–	Illegal Code	Causes illegal instruction interrupt to address 241_8	–	–
10	0-17	L, LA	Load A	(U)→A	.75	1.5
11	0-17	LN, LNA	Load Negative A	−(U)→A	.75	1.5
12	0-17	LM, LMA	Load Magnitude A	|(U)|→A	.75	1.5
13	0-17	LNMA	Load Negative Magnitude A	−|(U)|→A	.75	1.5
14	0-17	A, AA	Add To A	(A)+(U)→A	.75	1.5
15	0-17	AN, ANA	Add Negative To A	(A)−(U)→A	.75	1.5
16	0-17	AM, AMA	Add Magnitude To A	(A)+|(U)|→A	.75	1.5
17	0-17	ANM, ANMA	Add Negative Magnitude to A	(A)−|(U)|→A	.75	1.5
20	0-17	AU	Add Upper	(A)+(U)→A+1	.75	1.5
21	0-17	ANU	Add Negative Upper	(A)−(U)→A+1	.75	1.5
22	0-15	BT	Block Transfer	(X_x+u)→X_a+u; repeat K times	2.25+1.5K always	3.5 + 3.0K always
23	0-17	L, LR	Load R	(U)→R_a	.75	1.5
24	0-17	A, AX	Add To X	(X_a)+(U)→X_a	.75	1.5
25	0-17	AN, ANX	Add Negative To X	(X_a)−(U)→X_a	.75	1.5
26	0-17	LXM	Load X Modifier	(U)→$X_{a_{17-0}}$; $X_{a_{35-18}}$ unchanged	.875	1.666
27	0-17	L, LX	Load X	(U)→X_a	.75	1.5
30	0-17	MI	Multiply Integer	(A)•(U)→A,A+1	2.375	3.666
31	0-17	MSI	Multiply Single Integer	(A)•(U)→A	2.375	3.666
32	0-17	MF	Multiply Fractional	(A).(U)→A,A+1	2.375	3.666
33	–	–	Illegal Code	Causes illegal instruction interrupt to address 241_8	–	–
34	0-17	DI	Divide Integer	(A,A+1)÷(U)→A; REMAINDER→A+1	10.125	13.950
35	0-17	DSF	Divide Single Fractional	(A)÷(U)→A+1	10.125	13.950

Table B-1. Instruction Repertoire (Part 1 of 8)

Function Code (Octal) f	Function Code (Octal) j	Mnemonic	Instruction	Description ②	1108 Execution Time in μsecs. ①	1106 Execution Time in μsecs. ⑥
36	0-17	DF	Divide Fractional	(A,A+1)÷(U)→A; REMAINDER→A+1	10.125	13.950
37	–	–	Illegal Code	Causes illegal instruction interrupt to address 241_8	–	–
40	0-17	OR	Logical OR	(A) OR (U)→A+1	.75	1.5
41	0-17	XOR	Logical Exclusive OR	(A) XOR (U)→A+1	.75	1.5
42	0-17	AND	Logical AND	(A) AND (U)→A+1	.75	1.5
43	0-17	MLU	Masked Load Upper	[(U) AND (R2)] OR [(A) AND $(R2)^{'}$]→A+1	.75	1.5
44	0-17	TEP	Test Even Parity	Skip NI if (U) AND (A) have even parity	2.00 skip 1.25 NI	3.00 skip 2.166 NI
45	0-17	TOP	Test Odd Parity	Skip NI if (U) AND (A) have odd parity	2.00 skip 1.25 NI	3.00 skip 2.166 NI
46	0-17	LXI	Load X Increment	$(U)\rightarrow X_{a_{35-18}}$; $X_{a_{17-0}}$ unchanged	1.00	1.833
47	0-17	TLEM	Test Less Than or Equal To Modifier	Skip NI if $(U)\leq(X_a)_{17-0}$; always $(X_a)_{17-0}+(X_a)_{35-18}\rightarrow X_{a_{17-0}}$	1.75 skip 1.00 NI	3.333 skip 1.833 NI
		TNGM	Test Not Greater Than Modifier			
50	0-17	TZ	Test Zero	Skip NI if (U)=±0	1.625 skip .875 NI	3.166 skip 1.666 NI
51	0-17	TNZ	Test Nonzero	Skip NI if (U)≠±0	1.625 skip .875 NI	3.166 skip 1.666 NI
52	0-17	TE	Test Equal	Skip NI if (U)=(A)	1.625 skip .875 NI	3.166 skip 1.666 NI
53	0-17	TNE	Test Not Equal	Skip NI if (U)≠(A)	1.625 skip .875 NI	3.166 skip 1.666 NI
54	0-17	TLE TNG	Test Less Than or Equal Test Not Greater	Skip NI if (U)≤(A)	1.625 skip .875 NI	3.166 skip 1.66 NI
55	0-17	TG	Test Greater	Skip NI if (U)<(A)	1.625 skip .875 NI	3.166 skip 1.66 NI
56	0-17	TW	Test Within Range	Skip NI if (A)<(U)≤(A+1)	1.75 Skip 1.00 NI	3.33 skip 1.66 NI
57	0-17	TNW	Test Not Within Range	Skip NI if (U)≤(A) or (U)>(A+1)	1.75 skip 1.00 NI	3.33 skip 1.66 NI
60	0-17	TP	Test Positive	Skip NI if $(U)_{35}=0$	1.50 skip .75 NI	3.0 skip 1.5 NI
61	0-17	TN	Test Negative	Skip NI if $(U)_{35}=1$	1.50 skip .75 NI	3.0 skip 1.5 NI
62	0-17	SE	Search Equal	Skip NI if (U)=(A), else repeat	2.25 + .75K always	3.5 + 1.5K always
63	0-17	SNE	Search Not Equal	Skip NI if (U)≠(A), else repeat	2.25 + .75K always	3.5 + 1.5K always
64	0-17	SLE SNG	Search Less Than or Equal Search Not Greater	Skip NI if (U)≤(A), else repeat	2.25 + .75K always	3.5 + 1.5K always
65	0-17	SG	Search Greater	Skip NI if (U)>(A), else repeat	2.25 + .75K always	3.5 + 1.5K always

Table B-1. Instruction Repertoire (Part 2 of 8)

Function Code (Octal) f	Function Code (Octal) j	Mnemonic	Instruction	Description ②	1108 Execution Time in μsecs. ①	1106 Execution Time in μsecs. ⑥
66	0-17	SW	Search Within Range	Skip NI if (A)<(U)≤(A+1), else repeat	2.25 + .75K always	3.5 + 1.5K always
67	0-17	SNW	Search Not Within Range	Skip NI if (U)≤(A) or (U)>(A+1), else repeat	2.25 + .75K always	3.5 + 1.5K always
70	③	JGD	Jump Greater and Decrement	Jump to U if (Control Register)$_{ja}$>0; go to NI if (Control Register)$_{ja}$ ≤ 0; always (Control Register)$_{ja}$−1→ Control Register$_{ja}$	1.50 jump .75 NI	3.0 jump 1.5 NI
71	00	MSE	Mask Search Equal	Skip NI if (U) AND (R2)=(A) AND (R2), else repeat	2.25 + .75K always	3.5 + 1.5K always
71	01	MSNE	Mask Search Not Equal	Skip NI if (U) AND (R2)≠(A) AND (R2), else repeat	2.25 + .75K always	3.5 + 1.5K always
71	02	MSLE MSNG	Mask Search Less Than or Equal Mask Search Not Greater	Skip NI if (U) AND (R2)≤(A) AND (R2), else repeat	2.25 + .75K always	3.5 + 1.5K always
71	03	MSG	Mask Search Greater	Skip NI if (U) AND (R2)>(A) AND (R2), else repeat	2.25 + .75K always	3.5 + 1.5K always
71	04	MSW	Masked Search Within Range	Skip NI if (A) AND (R2)<(U) AND (R2)≤(A+1) AND (R2), else repeat	2.25 + .75K always	3.5 + 1.5K always
71	05	MSNW	Masked Search Not Within Range	Skip NI if (U) AND (R2)≤(A) AND (R2) or (U) AND (R2)>(A+1) AND (R2), else repeat	2.25 + .75K always	3.5 + 1.5K always
71	06	MASL	Masked Alphanumeric Search Less Than or Greater	Skip NI if (U) AND (R2)≤(A) AND (R2), else repeat	2.25 + .75K always	3.5 + 1.5K always
71	07	MASG	Masked Alphanumeric Search Greater	Skip NI if (U) AND (R2)>(A) AND (R2), else repeat	2.25 + .75K always	3.5 + 1.5K always
71	10	DA	Double Precision Fixed-Point Add	(A,A+1)+(U,U+1)→A,A+1	1.625	3.167
71	11	DAN	Double Precision Fixed-Point Add Negative	(A,A+1)−(U,U+1)→A,A+1	1.625	3.167
71	12	DS	Double Store A	(A,A+1)→U,U+1	1.50	3.0
71	13	DL	Double Load A	(U,U+1)→A,A+1	1.50	3.0
71	14	DLN	Double Load Negative A	−(U,U+1)→A,A+1	1.50	3.0
71	15	DLM	Double Load Magnitude A	|(U,U+1)|→A,A+1	1.50	3.0
71	16	DJZ	Double Precision Jump Zero	Jump to U if (A,A+1)=±0; go to NI if (A,A+1)≠±0	1.625 jump .875 NI	3.167 jump 1.667 NI
71	17	DTE	Double Precision Test Equal	Skip NI if (U,U+1)=(A,A+1)	2.375 skip 1.625 NI	4.667 skip 3.167 NI
72	00	–	Illegal Code	Causes illegal instruction interrupt to address 241_8	–	–
72	01	SLJ	Store Location and Jump	(P)−BASE ADDRESS MODIFIER [BI or BD]→U$_{17-0}$; jump to U+1	2.125 always	3.83

Table B-1. Instruction Repertoire (Part 3 of 8)

Function Code (Octal) f	Function Code (Octal) j	Mnemonic	Instruction	Description ②	1108 Execution Time in μsecs. ①	1106 Execution Time in μsecs. ⑥
72	02	JPS	Jump Positive and Shift	Jump to U if $(A)_{35}=0$; go to NI if $(A)_{35}=1$; always shift (A) left circularly one bit position.	1.50 jump .75 NI always	3.0 jump 1.5 NI always
72	03	JNS	Jump Negative and Shift	Jump to U if $(A)_{35}=1$; go to NI if $(A)_{35}=0$; always shift (A) left circularly one bit position	1.50 jump .75 NI always	3.0 jump 1.5 NI always
72	04	AH	Add Halves	$(A)_{35-18}+(U)_{35-18}\rightarrow A_{35-18}$; $(A)_{17-0}+(U)_{17-0}\rightarrow A_{17-0}$	.75 always	1.5 always
72	05	ANH	Add Negative Halves	$(A)_{35-18}-(U)_{35-18}\rightarrow A_{35-18}$; $(A)_{17-0}-(U)_{17-0}\rightarrow A_{17-0}$	.75 always	1.5 always
72	06	AT	Add Thirds	$(A)_{35-24}+(U)_{35-24}\rightarrow A_{35-24}$; $(A)_{23-12}+(U)_{23-12}\rightarrow A_{23-12}$; $(A)_{11-0}+(U)_{11-0}\rightarrow A_{11-0}$	.75 always	1.5 always
72	07	ANT	Add Negative Thirds	$(A)_{35-24}-(U)_{35-24}\rightarrow A_{35-24}$; $(A)_{23-12}-(U)_{23-12}\rightarrow A_{23-12}$; $(A)_{11-0}-(U)_{11-0}\rightarrow A_{11-0}$	.75 always	1.5 always
72	10	EX	Execute	Execute the instruction at U	.75 always	1.5 always
72	11	ER	Execute Return	Causes executive return interrupt to address 242_8	1.375 always	2.33 always
72	12	–	Illegal Code	Causes illegal instruction interrupt to address 241_8	–	–
72	13	PAIJ	Prevent All I/O Interrupts and Jump	Prevent all I/O interrupts and jump to U	.75 always	1.5 always
72	14	SCN	Store Channel Number	If a=0: CHANNEL NUMBER$\rightarrow U_{3-0}$; If a=1: CHANNEL NUMBER$\rightarrow U_{3-0}$ and CPU NUMBER$\rightarrow U_{5-4}$	.75	1.5
72	15	LPS	Load Processor State Register	(U)↦Processor State Register	.75	1.5
72	16	LSL	Load Storage Limits Register	(U)↦SLR	.75	1.5
72	17	–	Illegal Code	Causes illegal instruction interrupt to address 241_8	–	–
73	00	SSC	Single Shift Circular	Shift (A) right circularly U places	.75 always	1.5 always
73	01	DSC	Double Shift Circular	Shift (A,A+1) right circularly U places	.875 always	1.5 always
73	02	SSL	Single Shift Logical	Shift (A) right U places; zerofill	.75 always	1.5 always
73	03	DSL	Double Shift Logical	Shift (A,A+1) right U places; zerofill	.875 always	1.5 always
73	04	SSA	Single Shift Algebraic	Shift (A) right U places; signfill	.75 always	1.5 always
73	05	DSA	Double Shift Algebraic	Shift (A,A+1) right U places; signfill	.875 always	1.666 always
73	06	LSC	Load Shift and Count	(U)↦A, shift (A) left circularly until $(A)_{35}\neq(A)_{34}$; NUMBER OF SHIFTS $\rightarrow$A+1	1.125	2.0

Table B-1. Instruction Repertoire (Part 4 of 8)

Function Code (Octal) f	Function Code (Octal) j	Mnemonic	Instruction	Description ②	1108 Execution Time in μsecs. ①	1106 Execution Time in μsecs. ⑥
73	07	DLSC	Double Load Shift and Count	(U,U+1)→A,A+1; shift (A,A+1) left circularly until $(A,A+1)_{71} \neq (A,A+1)_{70}$; NUMBER OF SHIFTS→A+2	2.125	3.830
73	10	LSSC	Left Single Shift Circular	Shift (A) left circularly U places	.75 always	1.5 always
73	11	LDSC	Left Double Shift Circular	Shift (A,A+1) left circularly U places	.875 always	1.666 always
73	12	LSSL	Left Single Shift Logical	Shift (A) left U places; zerofill	.75 always	1.5 always
73	13	LDSL	Left Double Shift Logical	Shift (A,A+1) left U places; zerofill	.875 always	1.666 always
73	14	III (a=0 or 1)	Initiate Interprocessor Interrupt (1108 System only)	Initiate interprocessor interrupt	.75 always	–
		ALRM ($a=10_8$)	Alarm	Turn on alarm	.75 always	1.5 always
		EDC ($a=11_8$)	Enable Day Clock	Enable day clock	.75 always	1.5 always
		DDC ($a=12_8$)	Disable Day Clock	Disable day clock	.75 always	1.5 always
73	15	SIL	Select Interrupt Locations	(a)→MSR	.75 always	1.5 always
73	16	LCR (a=0)	Load Channel Select Register	$(U)_{3-0}$→CSR	.875	1.666
		LLA (a=1)	Load Last Address Register	$(U)_{2-0}$→LAR	.875	1.666
73	17	TS	Test and Set	If $(U)_{30}=1$, interrupt to address 244_8; if $(U)_{30}=0$, go to NI; always 01_8→U_{35-30}; $(U)_{29-0}$ unchanged	Alternate bank: 1.625 interrupt .875 NI	3.166
					Same bank: 2.0 interrupt 2.0 NI	1.666
74	00	JZ	Jump Zero	Jump to U if (A)=±0; go to NI if (A)≠±0	1.50 jump .75 NI always	3.0 jump 1.5 NI always
74	01	JNZ	Jump Nonzero	Jump to U if (A)≠±0; go to NI if (A)=±0	1.50 jump .75 NI always	3.0 jump 1.5 NI always
74	02	JP	Jump Positive	Jump to U if $(A)_{35}=0$; go to NI if $(A)_{35}=1$	1.50 jump .75 NI always	3.0 jump 1.5 NI always
74	03	JN	Jump Negative	Jump to U if $(A)_{35}=1$; go to NI if $(A)_{35}=0$	1.50 jump .75 NI always	3.0 jump 1.5 NI always
74	04	JK J	Jump Keys Jump	Jump to U if a=0 or if a=lit select jump indicator; go to NI if neither is true	.75 always	1.5 always
74	05	HKJ HJ	Half Keys and Jump Half Jump	Stop if a=0 or if [a AND lit select stop indicators]≠0; on restart or continuation, jump to U	.75 always	1.5 always

Table B-1. Instruction Repertoire (Part 5 of 8)

Function Code (Octal) f	Function Code (Octal) j	Mnemonic	Instruction	Description ②	1108 Execution Time in μsecs. ①	1106 Execution Time in μsecs. ⑥
74	06	NOP	No Operation	Proceed to next instruction	.75 always	1.5 always
74	07	AAIJ	Allow All I/O Interrupts	Allow all I/O interrupts and jump to U	.75 always	1.5 always
74	10	JNB	Jump No Low Bit	Jump to U if $(A)_0=0$; go to NI if $(A)_0=1$	1.50 jump .75 NI always	3.0 jump 1.5 NI always
74	11	JB	Jump Low Bit	Jump to U if $(A)_0=1$; go to NI if $(A)_0=0$	1.50 jump .75 NI always	3.0 jump 1.5 NI always
74	12	JMGI	Jump Modifier Greater and Increment	Jump to U if $(X_a)_{17-0}>0$; go to NI if $(X_a)_{17-0}\leq 0$; always $(X_a)_{17-0}+(X_a)_{35-18}\rightarrow X_{a\ 17-0}$	1.50 jump .75 NI always	3.166 jump 1.5 NI always
74	13	LMJ	Load Modifier and Jump	(P)−BASE ADDRESS MODIFIER [BI or BD]$\rightarrow X_{a\ 17-0}$; Jump to U	.875 always	1.666 always
74	14	JO	Jump Overflow	Jump to U if D1 of PSR=1; go to NI if D1=0	1.50 jump .75 NI always	3.0 jump 1.5 NI always
74	15	JNO	Jump No Overflow	Jump to U if D1 of PSR=0; go to NI if D1=1	1.50 jump .75 NI always	3.0 jump 1.5 NI always
74	16	JC	Jump Carry	Jump to U if D0 of PSR=1; go to NI if D0=0	1.50 jump .75 NI always	3.0 jump 1.5 NI always
74	17	JNC	Jump No Carry	Jump to U if D0 of PSR=0; go to NI if D0=1	1.50 jump .75 NI always	3.0 jump 1.5 NI always
75	00	LIC	Load Input Channel	For channel [a OR CSR]:(U)→IACR; set input active; clear input monitor	.75	1.5
75	01	LICM	Load Input Channel and Monitor	For channel [a OR CSR]:(U)→IACR; set input active; set input monitor	.75	1.5
75	02	JIC	Jump On Input Channel Busy	Jump to U if input active is set for channel [a OR CSR]; go to NI if input active is clear	.75 always	1.5 always
75	03	DIC	Disconnect Input Channel	For channel [a OR CSR]: clear input active; clear input monitor	.75 always	1.5 always
75	04	LOC	Load Output Channel	For channel [a OR CSR]:(U)→OACR; set output active; clear output monitor; clear external monitor (ISI only)	.75	1.5 always
75	05	LOCM	Load Output Channel and Monitor	For channel [a OR CSR]:(U)→OACR; set output active; set output monitor; clear external function (ISI only)	.75	1.5
75	06	JOC	Jump On Output Channel Busy	Jump to U if output active is set for channel [a OR CSR]; go to NI if output active is clear	.75 always	1.5 always

Table B-1. Instruction Repertoire (Part 6 of 8)

Function Code (Octal) f	Function Code (Octal) j	Mnemonic	Instruction	Description ②	1108 Execution Time in μsecs. ①	1106 Execution Time in μsecs. ⑥
75	07	DOC	Disconnect Output Channel	For channel [a OR CSR]: clear output active; clear output monitor; clear external function	.75 always	1.5 always
75	10	LFC	Load Function in Channel	For channel [a OR CSR]: (U)→ OACR; set output active (ISI only), external function, and force external function; clear output monitor (ISI only)	.75	1.5
75	11	LFCM	Load Function in Channel and Monitor	For channel [a OR CSR]: (U)→ OACR; set output active (ISI only), external function, force external function, and output monitor (ISI only)	.75	1.5
75	12	JFC	Jump On Function in Channel	Jump to U if force external function is set for channel [a OR CSR]; go to NI if force external function is clear	.75 always	1.5 always
75	13	–	Illegal Code	If guard mode is set, causes guard mode interrupt to address 243_8. If guard mode is not set, same as NOP	.75 always	1.5 always
75	14	AACI	Allow All Channel External Interrupts	Allow all external interrupts	.75 always	1.5 always
75	15	PACI	Prevent All Channel External Interrupts	Prevent all external interrupts	.75 always	1.5 always
75	16	–	Illegal Code	If guard mode is set, causes guard mode interrupt to address 243_8. If guard mode is not set, same as NOP	.75 always	1.5 always
75	17	–	Illegal Code			
76	00	FA	Floating Add	(A)+(U)→A; RESIDUE→A+1	1.875	3.0
76	01	FAN	Floating Add Negative	(A)−(U)→A; RESIDUE→A+1	1.875	3.0
76	02	FM	Floating Multiply	(A)·(U)→A,A+1	2.625	4.0
76	03	FD	Floating Divide	(A)÷(U)→A; REMAINDER→A+1	8.25 ④	11.5
76	04	LUF	Load and Unpack Floating	$\lvert(U)\rvert_{34-27}\rightarrow A_{7-0}$, zerofill; $(U)_{26-0}\rightarrow A+1_{26-0}$, signfill	.75 always	1.5 always
76	05	LCF	Load and Convert To Floating	$(U)_{35}\rightarrow A+1_{35}$; [NORMALIZED (U)]$_{26-0}$ $\rightarrow A+1_{26-0}$; if $(U)_{35}=0$, $(A)_{7-0}\pm$ NORMALIZING COUNT$\rightarrow A+1_{34-27}$; if $(U)_{35}=1$, ones complement of $[(A)_{7-0}\pm$NORMALIZING COUNT$]\rightarrow A+1_{34-27}$	1.125	2.0
76	06	MCDU	Magnitude of Characteristic Difference To Upper	$\lvert\lvert(A)\rvert_{35-27}-\lvert(U)\rvert_{35-27}\rvert\rightarrow A+1_{8-0}$; ZEROS$\rightarrow A+1_{35-9}$	.75 always	1.5 always
76	07	CDU	Characteristic Difference To Upper	$\lvert(A)\rvert_{35-27}-\lvert(U)\rvert_{35-27}\rightarrow A+1_{8-0}$; SIGN BITS$\rightarrow A+1_{35-9}$	.75 always	1.5 always
76	10	DFA	Double Precision Floating Add	(A,A+1)+(U,U+1)→A,A+1	2.625	4.5

Table B-1. Instruction Repertoire (Part 7 of 8)

Function Code (Octal) f	Function Code (Octal) j	Mnemonic	Instruction	Description ②	1108 Execution Time in μsecs. ①	1106 Execution Time in μsecs. ⑥
76	11	DFAN	Double Precision Floating Add Negative	$(A,A+1)-(U,U+1)\rightarrow A,A+1$	2.625	4.5
76	12	DFM	Double Precision Floating Multiply	$(A,A+1)\cdot(U,U+1)\rightarrow A,A+1$	4.25	6.667
76	13	DFD	Double Precision Floating Divide	$(A,A+1)\div(U,U+1)\rightarrow A,A+1$	17.25 ⑤	24.0 ⑧
76	14	DFU	Double Load and Unpack Floating	$\lvert(U)\rvert_{34-24}\rightarrow A_{10-0}$, zerofill; $(U)_{23-0}\rightarrow A+1_{23-0}$, signfill; $(U+1)\rightarrow A+2$	1.50	3.0
76	15	DEP	Double Load and Convert To Floating	$(U)_{35}\rightarrow A+1_{35}$; [NORMALIZED $(U,U+1)]_{59-0}\rightarrow A+1_{23-0}$ and A+2; if $(U)_{35}=0$, $(A)_{10-0}\pm$NORMALIZING COUNT$\rightarrow A+1_{34-24}$; if $(U)_{35}=1$, ones complement of $[(A)_{10-0}\pm$ NORMALIZING COUNT$]\rightarrow A+1_{34-24}$		
76	16	FEL	Floating Expand and Load	If $(U)_{35}=0$, $(U)_{35-27}+1600_8\rightarrow A_{35-24}$; if $(U)_{35}=1$, $(U)_{35-27}-1600_8\rightarrow A_{35-24}$; $(U)_{26-3}\rightarrow A_{23-0}$; $(U)_{2-0}\rightarrow A+1_{35-33}$; $(U)_{35}\rightarrow A+1_{32-0}$	1.00	1.833
76	17	FCL	Floating Compress and Load	If $(U)_{35}=0$, $(U)_{35-24}-1600_8\rightarrow A_{35-27}$; if $(U)_{35}=1$, $(U)_{35-24}+1600_8\rightarrow A_{35-27}$; $(U)_{23-0}\rightarrow A_{26-3}$; $(U+1)_{35-33}\rightarrow A_{2-0}$	1.625	3.167
77	0-17	–	Illegal Code	Causes illegal instruction interrupt to address 241_8	–	–

Table B-1. Instruction Repertoire (Part 8 of 8)

NOTES:

① The execution times given are for alternate bank memory access; for same bank memory access, execution time is .75 microseconds greater. Exceptions to this either show the execution times for both types of memory access or include the word "always" to indicate that the execution time is the same regardless of the type of memory access.

For function codes 01 through 06 and 22, add .375 microseconds to the execution times for 6-bit and 12-bit writes.

The execution time for a Block Transfer or any of the search instructions depends on the number of repetitions (K) required; that is, the number of words in the block being transferred or the number of words searched before a find is made.

② NI stands for Next Instruction.

③ The a and j fields together serve to specify any of the 128 control registers.

④ If 28 rather than 27 subtractions are performed, add .25 microseconds to the execution time.

⑤ If 61 rather than 60 subtractions are performed, add .25 microseconds to the execution time.

⑥ Execution times given are calculated using a main storage cycle time of 1.5 microseconds and a CPU clock cycle time of 166 nanoseconds.

For all comparison instructions, the first number represents the skip or jump condition, the second number is for a no skip or no jump condition.

For function codes 01 through 67, add .333 microseconds to execution times for 6-bit, 9-bit, and 12-bit writes.

Execution time for the Block Transfer and the search instructions depends on the number of repetitions of the instruction required. The variance is 3.0K microseconds for Block Transfer and 1.5K microseconds for searches where K equals the number of repetitions; that is, K equals the number of words in the block being transferred or the number of words searched before a match is found.

⑦ If 28 instead of 27 subtractions are performed, add .333 microseconds.

⑧ If 61 instead of 60 subtractions are performed, add .333 microseconds.

Mnemonic	1106/1108 Function Code (Octal) f	i	Mnemonic	1106/1108 Function Code (Octal) f	i	Menomic	1106/1108 Function Code (Octal) f	i	Menomic	1106/1108 Function Code (Octal) f	i
A	14	0-17	DF	36	0-17	FEL	76	16	LCF	76	05
A	24	0-17	DFA	76	10	FM	76	02	LCR	73 a=0	16
AA	14	0-17	DFAN	76	11	HJ	74	05			
AACI	75	14	DFD	76	13	HKJ	74	05	LDSC	73	11
AAIJ	74	07	DFM	76	12	III	73 a=0 or 1	14	LDSL	73	13
AH	72	04	DFP	76	15				LCF	75	10
ALRM	73 $a=10_8$	14	DFU	76	14	J	74	09	LFCM	75	11
			DI	34	0-17	JB	74	11	LIC	75	00
AM	16	0-17	DIC	75	03	JC	74	16	LICM	75	01
AMA	16	0-17	DJZ	71	16	JFC	75	12	LLA	73 a=1	16
AN	15	0-17	DL	71	13	JGD	70	†			
AN	25	0-17	DLM	71	15	JIC	75	02	LM	12	0-17
ANA	15	0-17	DLN	71	14	JK	74	04	LMA	12	0-17
AND	42	0-17	DLSC	73	07	JMGI	74	12	LMJ	74	13
ANH	72	05	DOC	75	07	JN	74	03	LN	11	0-17
ANM	17	0-17	DS	71	12	JNB	74	10	LNA	11	0-17
ANMA	17	0-17	DSA	73	05	JNC	74	17	LNMA	13	0-17
ANT	72	07	DSC	73	01	JNO	74	15	LOC	75	04
ANU	21	0-17	DSF	35	0-17	JNS	72	03	LOCM	75	05
ANX	25	0-17	DSL	75	03	JNZ	74	01	LPS	72	15
AT	72	06	DTE	71	17	JO	74	14	LR	23	0-17
AU	20	0-17	EDC	73 $a=11_8$	14	JOC	75	06	LSC	73	06
AX	24	0-17				JP	74	02	LSL	72	16
BT	22	0-15	ER	72	11	JPS	72	02	LSSC	73	10
CDU	76	07	EX	72	10	JZ	74	00	LSSL	73	12
DA	71	10	FA	76	00	L	10	0-17	LUF	76	04
DAN	71	11	FAN	76	01	L	23	0-17	LX	27	0-17
DDC	73 $a=12_8$	14	FCL	76	17	L	27	0-17	LXI	46	0-17
			FD	76	03	LA	10	0-17	LXM	26	0-17
MASG	71	07	S	01	0-17	SSA	73	04	TOP	45	0-17
MASL	71	06	S	04	0-17	SSC	73	00	TP	60	0-17
MCDU	76	06	S	06	0-17	SSL	73	02	TS	73	17
MF	32	0-17	SA	01	0-15	SW	66	0-17	TW	56	0-17
MI	30	0-17	SCN	72	14	SX	06	0-15	TZ	50	0-17
MLU	43	0-17	SE	62	0-17	SZ	05	0-15	XOR	41	0-17
MSE	71	00	SG	65	0-17	TE	52	0-17	–	00	
MSG	71	03	SIL	73	15	TEP	44	0-17	–	07	
MSI	31	0-17	SLE	64	0-15	TG	55	0-17	–	33	
MSLE	71	02	SLJ	72	01	TLE	54	0-17	–	37	
MSNE	71	01	SM	03	0-15	TLEM	47	0-17	–	72	00
MSNG	71	02	SMA	03	0-15	TN	61	0-17	–	72	12
MSNW	71	05	SN	02	0-15	TNE	53	0-17	–	72	17
MSW	71	04	SNA	02	0-15	TNG	54	0-17	–	75	13
NOP	74	06	SNE	63	0-17	TNGM	47	0-17	–	75	16
OR	40	0-17	SNG	64	0-17	TNW	57	0-17	–	75	17
PACI	75	15	SNW	67	0-17	TNZ	51	0-17	–	77	
PAIJ	72	13	SR	04	0-17						

† *The j and a fields together serve to specify any of the 128 control registers.*

Table B-2. Mnemonic/Function Code Cross-Reference

APPENDIX C. ASSEMBLER ERROR FLAGS AND MESSAGES

C.1. ERROR FLAGS

C.1.1. R-Relocation

An R flag indicates that a relocatable item (usually a label) has been so used in an expression as to cause loss of its relocation properties. Appendix D shows the results of all combinations of relocatable and nonrelocatable items.

C.1.2. E-Expression

Expression error flags may be produced in a variety of ways, such as the inclusion of a decimal digit in an octal number (for example, 080), and binary or decimal exponentiation with a real exponent (for example, 3.14*/1.2).

C.1.3. T-Truncation

The T flag indicates that a value is too large for its destined field. Consider the following example:

```
F           FORM    18,18
A    EQU    +(F     0,-3)     (1)
G           FORM    32,4
            G       0,A       (2)
```

The form reference in line (1) is legitimate, but (2) would produce a T flag, since the value of A in this case is 000000777774_8 (a value with 18 significant bits), and the second field of form ''G'' is defined as four bits in length.

The T flag will also appear on a line containing a location counter reference greater than 31 (37_8, or 5 bits).

C.1.4. L-Level

This flag indicates that some capacity of the assembler, such as a table count, has been exceeded. The limits listed below are generous; but if one is exceeded, simplification of coding is required.

(a) Nested procedure or function references may not be more than 62 deep.

(b) Parentheses nests, including nested literals, may not be more than 8 deep; this includes parentheses used for grouping of terms.

(c) Nested DO's may not be more than 8 deep.

C.1.5. D-Duplicate

Labels, disregarding possible subscripts, must be unique in a given assembly or subassembly. Redefinition of a label produces a D flag on each line in which the label appears, unless the label is subscripted. The obvious mistake

```
A       EQU     1
        .
        .
        .
A       EQU     2
```

is easily discovered. Much more insidious is the redefinition in assembly pass 2 of a label previously assigned a different value in pass 1. This usually results from an illegal manipulation of a location counter.

C.1.6. I-Instruction

If the first subfield in the operation field of a symbolic line contains neither the name of a directive, nor an available procedure, nor a FORM reference, nor a mnemonic, an I flag is produced. A procedure is considered available only if it is in the procedure library (that is, the system relocatable library or a user's file), or if it has previously been encountered in the source program. See Appendix F for rules governing the searching of procedures.

C.1.7. U-Undefined

If an operand symbol is not defined in the source program, each line containing the symbol is marked – with a U flag – as containing an undefined symbol. In some cases, this may denote a reference to a value externally defined in some other independently processed code. But there is the chance that a U flag might simply denote an error by the programmer.

C.2. ERROR MESSAGES

1108 ASM Internal Error Abort

The assembler has lost control of what it is doing. This may result from nearly any cause including an anomaly in the assembler or executive system, or an undetected data transmission error. Index register X11 contains the location at which the error was detected. The assembly is terminated in error.

Abort Cannot Read PROC from Drum

An I/O error resulted when the assembler attempted to read a procedure from a drum or FASTRAND file. The assembly is terminated in error.

Assembler Image

An end of file was detected on the source file. An END card with the above comment is supplied. Processing terminates normally, but the element is marked as being in error.

ASM Abort no Scratch File A0 XXXXX

The assembler is unable to dynamically assign a scratch file. The A0 value indicated is the status word returned by the executive system. For meaning of the status word, see *UNIVAC 1108 Multi-Processor System Executive Programmers Reference Manual, UP-4144* (current version). The assembly is terminated in error.

Bad Procedure Read

An I/O error was detected in attempting to read a procedure sample from mass storage. Processing continues by searching next mass storage procedure file.

Item Table Overflow

Insufficient space exists for the assembler to define a symbol or literal. The assembly is terminated in error.

Line Number Sequence Errors

The symbolic corrections inserted as input to this assembly are out of sequence. The assembly continues. Source lines following the out-of-sequence correction card will be inserted at the point at which the error is detected.

PARTBL Not Initialized

The preprocessor routine is unable to initialize the assembler parameter table. Probable causes are incorrect file assignments, incorrect processor control card, or I/O error. The assembly is terminated in error. The preprocessor also prints a message indicating the nature of the error.

Procedure Sample Storage Overflow

Insufficient space exists for the assembler to process a line of procedure definitions. The assembly is terminated in error.

ROR Internal Error Abort

The relocatable output routine is unable to write a record of relocatable binary output probably because of an I/O error or improper file assignment. The assembly is terminated in error.

TBLWR$ Internal Error Abort

The relocatable output routine is unable to write the preamble to the relocatable output file (probably because of an I/O error). The assembly is terminated in error.

APPENDIX D. RULES OF OPERATORS

D.1. RULES FOR DETERMINING RESULTS OF OPERATIONS

LEVEL	1st ITEM	OP	2nd ITEM	RESULT
6	Any	*+	Binary①	Positive Decimal Exponentiation
	Any	*–	Binary①	Negative Decimal Exponentiation
	Any	*/	Positive Binary①	Positive Binary Exponentiation
	Any	*/	Negative Binary①	Negative Binary Exponentiation Sign filled
5	Any	*	Any	Arithmetic product
	Any	/	Any	Arithmetic quotient
	Any	//	Any	Arithmetic covered quotient
4	Any	+	Any	Arithmetic sum
	Any	–	Any	Arithmetic difference
3	Any	**	Any	Logical product
2	Any	++	Any	Logical sum
		––	Any	Logical difference
1	Any	<,=,>	Any	1 if true 0 if false

① A nonbinary, that is, floating-point value results in an expression error flag (E).

D.2. RULES FOR DETERMINING MODES OF RESULTS

LEVEL	1st ITEM	OP	2nd ITEM	RESULT
6	Any	*+, *–	Binary①	Floating
	Any	*/	Binary①	Binary
5	Binary	*,/,//	Binary	Binary
	Floating	*,/,//	Binary	Floating
	Binary	*,/,//	Floating	Floating
	Floating	*,/,//	Floating	Floating
4	Binary	+,–	Binary	Binary
	Floating	+,–	Binary	Floating
	Binary	+,–	Floating	Floating
	Floating	+,–	Floating	Floating
3	Any	**	Any	Binary
2	Any	++, ––	Any	Binary
1	Any	<, =, >	Any	Binary

① A nonbinary, that is, floating-point value results in an expression error flag (E).

D.3. RULES FOR RELOCATION OF BINARY ITEMS

LEVEL	1st ITEM	OP	2nd ITEM	RESULT	NOTE
1	Any	<, =, >	Any	Not relocatable	
2	Any	++, −−	Any	Not relocatable	3
3	Any	**	Any	Not relocatable	3
4	Not relocatable	+, −	Not relocatable	Not relocatable	
	Relocatable	+, −	Not relocatable	Relocatable	
	Not relocatable	+, −	Relocatable	Relocatable	
	Relocatable	+, −	Relocatable	Relocatable	2
5	Any	*, /, //	Any	Not relocatable	3, 4
6	Any	*+, *−, */	Binary	Not relocatable	3, 5

① Floating-point items are never relocatable.

② The difference between two relocatable quantities under the same location counter is not relocatable.

③ Except as noted in 4, the relocation error flag (R) is set for these operations.

④ Multiplication of a relocatable quantity by an absolute 1, or absolute 1 by a relocatable quantity, is relocatable. Multiplication by absolute 0 is absolute 0. In either case, no error flag is set.

⑤ A nonbinary, that is, floating-point value for the 2nd item results in an expression error flag (E).

D.4. RULES FOR HANDLING SINGLE AND DOUBLE PRECISION EXPRESSIONS

OPERATION	FIRST VALUE	SECOND VALUE	RESULT
>, <, =	Single	Single	Single
		Double	
	Double	Single	
		Double	
+, −, ++, −−, **	Single	Single	Single ①
		Double	Double
	Double	Single	
		Double	
*, /, //	Single	Single	Single ①
		Double	Double ②
	Double	Single	
		Double	
*/, * +, *−	Single	Single	Single
		Double	
	Double	Single	Double
		Double	

NOTES:

① Multiplication, addition, or subtraction in fixed-point modes may result in a double precision value.

② These cases are not permitted for fixed-point values. If fixed point values are used, however, they result in a single precision result with an E error flag.

APPENDIX E. FORMAT OF ASM CONTROL CARD

The format of the assembly control card is:

@ASM,Options F1.E1, F2.E2, F3.E3

where options letters interrogated by the assembler are:

C Produce symbolic listing (no octal).
D Produce double-spaced listing.
L Produce complete listing (octal, symbolic, and relocation information).
M Request 10K additional main storage for symbol and procedure sample table.
N Suppress all listing.
O Produce octal listing only (no symbolic).
R Release 5K additional main storage for symbol and procedure sample table.
T Request 5K additional main storage for symbol and procedure sample table.

Other options applicable to an assembly are (interrogated by the source input routine, SIR):

U Update and produce new cycle of source element.
I Insert new element to program file from control stream.
W List corrections.

and

F1.E1 are the input source file and element.
F2.E2 are the relocatable file and element.
F3.E3 are the updated source file and element.

If the I option is selected (as when inserting from cards), specification field 1 names the program file to contain the source code. If assembling from tape, field 1 is the file name of that tape, field 2 is the relocatable program file name, and field 3 specifies the name of the program file to contain the source code. If file names are not specified, the temporary run file is utilized.

For the detailed description of the card format, file-element entries, and formats of correction lines, see the *UNIVAC 1108 Multi-Processor System Executive Programmers Reference Manual, UP-4144* (current version).

APPENDIX F. RULES FOR PROCEDURE SEARCHING

The assembler searches one user program file for procedures and if it fails to find the procedure, it searches the system relocatable library, RLIB$. If the user has more than one file, the order of precedence for searching is as follows:

1. If input is from a program file on drum as determined by character one of PARTBL = 050, the user source input file (file name in PARTBL + 1 and PARTBL + 2) is searched.

2. If character one of PARTBL ≠ 050, the assembler searches the source output file, file name in PARTBL + 14 and PARTBL + 15, provided a source output file has been specified as determined by PARTBL + 14 ≠ 'SCRFIL'.

3. If neither a drum source input or source output file has been specified as determined by character one of PARTBL ≠ 050 and PARTBL + 14 = 'SCRFIL', then the user relocatable output file is searched for the procedure. The relocatable output file name is taken from PARTBL + 27 and PARTBL + 28.

Note that a maximum of two files is searched, one user file and the system relocatable library.

APPENDIX G. CONSIDERATIONS FOR DEMAND PROCESSING

The assembler does not operate in the demand (conversational) mode as such (EXEC 8 operation only); instead provision is made to operate the assembler from remote terminals in the batch mode. For a description of initiation of runs from demand terminals, see *UNIVAC 1108 Multi-Processor System Executive System Programmers Reference Manual, UP-4144* (current version).

NOTE: Demand mode processing is not possible when using the EXEC II.

The assembler control language contains two option letters for controlling the output listing expressly for demand terminal use. These options are not required, but when used provide the capability to obtain abbreviated output listings of the assembled element.

C option (list symbolic only)

The C option generates a listing in the following format:

NNNNN EEEEE SYMBOLIC SOURCE LINE

where NNNNN is a five-digit line-sequence number.

EEEEE are error flags for errors detected by the assembler; they may all be blank.

SYMBOLIC SOURCE LINE is the input data line.

O option (list octal values only)

The listing produced by the O option has the following format:

LLLLLL EEEEE CC LOCATION VALUE

where LLLLLL is a 6-digit line-sequence number.

EEEEE are error flags for any errors detected by the assembler; they may all be blank.

CC is the location counter under which data is being generated.

LOCATION is the relative location in main storage under the indicated CC, for which the data is being generated.

VALUE is the object code generated by the assembler for the symbolic code.

APPENDIX H. 1106/1108 ASSEMBLER OPERATING UNDER ALTERNATE EXECUTIVE SYSTEMS

H.1. DIFFERENCES IN OPERATION

The assembler can be operated under control of the EXEC II operating system on either UNIVAC 1106, UNIVAC 1107, or UNIVAC 1108 Computer. The following summarizes the differences between the assembler operating under EXEC 8 and the assembler operating under EXEC II.

- All instruction addresses are considered to be 16 bits in length (see 1.2.2).

- The 1107 – EXEC II version does not contain any double precision data capability. The 1108 – EXEC II version does, however, contain the same double precision capability described in this manual.

- The EXEC II versions contain an automatic capability to obtain definitions of system symbols. The M option in the EXEC II versions can be used to suppress the definitions of system symbols. In EXEC 8, executive system symbols are made available only if expressly requested by placing appropriate procedure calls into the object program being assembled. See discussion of EXEC II options in the *UNIVAC 1106/1108 EXEC II Programmers Reference Manual, UP-4058* (current version).

- When operating under EXEC II, the method of searching mass storage for procedures referenced in the source program being assembled differs from the scheme outlined in Appendix F of this manual. First, the user PCF is searched for the procedure definition; if the procedure is not found, the library PCF is searched for the procedure.

- The format of the ASM control card differs from that used in EXEC 8; see *UNIVAC 1106/1108 EXEC II Programmers Reference Manual, UP-4058* (current version).

- Options letters and their meanings differ in the EXEC II versions; see *UNIVAC 1106/1108 EXEC II Programmers Reference Manual, UP-4058* (current version).

H.2. ERROR MESSAGES GENERATED BY THE EXEC II ASSEMBLY SYSTEM

***New Line Insert

The following line was taken from the stream of source corrections supplied with the assembly. This is not an error message.

BOP Internal Error Abort

An internal error has been detected by the Binary Output Package. The assembly is aborted.

Drum ERR Abort

An I/O error occurred while reading or writing mass storage. The assembly is aborted.

Drum Error On Drum Procedure Read

An I/O error occurred while reading procedure sample from mass storage. The assembly is aborted.

Item Table Overflow

Insufficient space remains to insert the next literal or symbol definition into the assembler's item table. The assembly is aborted.

Procedure Sample Storage Overflow

Insufficient space remains in the assembler sample storage area to contain a line of procedure definition. The assembly is aborted.

Line Deletion

A line of source code was deleted at this point in the program. This is not an error message.

Line Number Sequence Error

Correction lines are out of sequence. Any source lines following the erroneous correction are inserted at the place the error was detected. Processing continues.

Sleuth Card

An end of file condition was detected on the source input file. This comment appears in the comment field of an END card supplied by the assembler. Processing continues but the error exit will be used to terminate the assembly.

UNIVAC

SPERRY RAND

UP-4040 Rev. 2