SPERRY RAND

UNIVAC

418-III

REAL-TIME SYSTEM

RTOS ASSEMBLER

PROGRAMMERS REFERENCE

UP-7599 Rev. 1

CONTENTS

1. THE UNIVAC 418-III ASSEMBLER

1.1. INTRODUCTION

The UNIVAC 418-III Assembler 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 418-III System.

The assembler operates under control of the Real-Time Operating System (RTOS). The output of the assembler is made consistent with the system by using standard interfacing routines both for the source files and the relocatable program generated.

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 an edited octal representation of each word generated. Flags indicate errors 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: a label field, an operation field, and 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 element.

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. Assembler Character Set

The assembler uses the XS-3 character set as defined in Table 1–1. If alphanumeric data is to be generated in a different code convention, the CHAR directive, described in 2.7, may be used.

80 COL. CARD CODE	PRINTABLE CHARACTERS	XS-3 CODE	80 COL. CARD CODE	PRINTABLE CHARACTERS	XS-3 CODE
12-1	A	01 0100	7	7	00 1010
12-2	B	01 0101	8	8	00 1011
12-3	C	01 0110	9	9	00 1100
12-4	D	01 0111	12	+	01 0000
12-5	E	01 1000	11	–(Minus)	00 0010
12-6	F	01 1001	12-0	?	01 0011
12-7	G	01 1010	11-0	!(Exclam.)	10 0011
12-8	H	01 1011	0-1	/	11 0100
12-9	I	01 1100	2-8	&	11 0011
11-1	J	10 0100	3-8	=	01 1101
11-2	K	10 0101	4-8	'(Apos.)	10 1110
11-3	L	10 0110	5-8	:(Colon)	01 0001
11-4	M	10 0111	6-8	>	11 1110
11-5	N	10 1000	7-8	@	10 0000
11-6	O	10 1001	12-3-8	.(Period)	01 0010
11-7	P	10 1010	12-4-8	)	11 1101
11-8	Q	10 1011	12-5-8	[	00 1111
11-9	R	10 1100	12-6-8	<	01 1110
0-2	S	11 0101	12-7-8	#	01 1111
0-3	T	11 0110	11-3-8	$	10 0010
0-4	U	11 0111	11-4-8	*	10 0001
0-5	V	11 1000	11-5-8	]	00 0001
0-6	W	11 1001	11-6-8	;(Semi-col)	00 1110
0-7	X	11 1010	11-7-8	Δ	10 1111
0-8	Y	11 1011	0-2-8	≠	11 0000
0-9	Z	11 1100	0-3-8	,(Comma)	11 0010
0	0	00 0011	0-4-8	(	11 0001
1	1	00 0100	0-5-8	%	10 1101
2	2	00 0101	0-6-8	\	00 1101
3	3	00 0110	0-7-8	¤	11 1111
4	4	00 0111			
5	5	00 1000	BLANK	Space N.P.	00 0000
6	6	00 1001			

Table 1–1. Assembler Character Set

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.

1.3.1.1. Simple 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, LIT (see Section 2).

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

Special characters are not allowed within a label. To ensure uniqueness, many system labels use the $ as part of the label. Using the $ as part of a label should be avoided to assure this uniqueness of system labels.

Labels defined in the aforementioned manner are referred to as simple labels and are allowed on any statement. If a label is the only nonblank field on a statement, the label is defined as identifying the next location counter value to be generated.

Example:

```
   LABEL         OPERATION               OPERAND                   COMMENTS
1           10            20          30          40          50
LABEL      LLK       5
A12$6.
BCD        LL        A12$6
```

1.3.1.2. External Labels

An externally defined label is one which may be accessed by other programs. The loader will correlate the references between the external label references in one program and the corresponding external label definitions in another. To define an external label, an asterisk is appended to the label.

Example:

```
   LABEL         OPERATION               OPERAND                   COMMENTS
1           10            20          30          40          50
LABEL      LLK       5         .LOCAL LABEL
TAG*       LLK       5         .EXTERNALLY DEFINED LABEL
```

1.3.1.3. Dimensioned (Subscripted) Labels

A dimensioned or subscripted label is a label which is distinguished by its subscripts rather than by the label itself. The label serves to identify a set of related quantities. A subscript may be any legitimate assembler item, an expression, or another subscripted label. In defining a subscripted label, all symbols used in expressing any of its subscripts must have been previously defined. If another dimensioned label is used as a subscript of the label being defined, it must have been defined previously.

The dimensioned label is identified by the format:

label(sub_1, sub_2,...,sub_n)

The number of subscripts used in defining a dimensioned label is referred to as its dimensionality. The maximum dimensionality of a subscripted label is unspecified. The dimensionality of a subscripted label is constant, that is, once a member of the set is defined, all other explicitly defined members must have the same number of subscripts even though each subscript value may differ.

Example:

LABEL	OPERATION	OPERAND
A(3)	+5	
B(4,1)	+6	
C(A(3),2)	+7	
D(C(A(3),B(4,1)),1)	+8	

Explanation:

- Line 1 defines a one-dimensional label A(3). The subscript value is 3.
- Line 2 defines a two-dimensional label B(4,1) with subscript values 4 and 1.
- Line 3 defines a two-dimensional label C(A(3),2) with subscript values A(3) and 2.
- Line 4 defines a two-dimensional label D(C(A(3),B(4,1)),1) with subscript values C(A(3),B(4,1)) and 1.

Dimensioned labels may not be defined to be external to the program assembly. If used within procedures (see Section 3), the dimensioned labels may be defined as accessible at lower levels by appending the appropriate number of asterisks immediately following the label and before the left parenthesis.

Dimensioned labels may be defined to have a value in magnitude of 2^{36}-1 or less. If any item used in defining the value of the expression is a double-word item (see 1.4), the label has a double-word value (see 2.2).

The value of a dimensioned label may be redefined in the course of the assembly without resulting in a 'D-flag'.

If reference is made to an undefined member of a defined set of dimensioned labels, the value of the undefined item is assumed to be a defined zero. If no member of the set is defined, the value is zero and an external reference is made to the label.

Example:

```
000001                                              /.
000002   U      00 000000   70 0000                          LLK      A(1)
                              + 6,12 00     A
000003                      000001               B(1)        EQU      1
000004          00 000001   70 0000                          LLK      B(2)
                              + 6,12 00
000005                                                       END
                                                    *** SUMMARY ***

     PROGRAM SIZE:    00 00002

     EXTERNAL OR UNDEFINED REFERENCES:    A
```

Explanation:

- Line 2 references the label A(1). Since no member of the set A(i) is defined, an external reference to A is made.
- Line 3 defines the set B(i) in general and the member B(1) in particular.
- Line 4 references an undefined member of the set B(i). Its value is taken to be zero.

If reference is made to a dimensioned label, some member of which was previously defined with a smaller dimensionality, an expression error results, and the value of the referenced label is taken to be zero.

Example:

LABEL	OPERATION	OPERAND
A(1)	EQU	5
	LLK	A(1,2)

As stated previously, the dimensionality of a subscripted label is constant. As a result, all members of a set of dimensioned labels must have the same number of subscripts. An expression error results if a subscripted label is defined at a different dimensionality than another member of the same set, that is, with different subscript values but using the same label.

While the user defines the values of a particular dimensioned label, the assembler internally defines values for the label with lower dimensionalities. These may be referenced (but not defined) in the course of the assembly. For example, if the label A(1,2,3,4) is defined, the labels A, A(1), A(1,2), and A(1,2,3) are internally defined by the assembler. (Note that the name of a dimensioned label must be unique and may not duplicate a simple label.) If a dimensioned label is defined, all labels of lower dimensionality having the same name are therefore implicitly defined by the assembler. The values associated with these assembler-defined labels is described in the following paragraphs.

An n-dimensional set of labels, $A(s_1,s_2,s_3,\ldots,s_n)$ is defined. Many different values of each of the subscripts s_i may have been used in defining the set of labels. Each subscript s_i has been used n_i times; there are n_i different subscript values s_i.

The set of labels defined is:

A(1,2,3)
A(5,7,3)
A(5,8,3)
A(7,2,2)
A(8,9,0)
A(1,2,4)

Then: $n_1 = 4$ because there are four different subscript values defined in the first dimension;

for $s_1 = 1$, $n_2 = 1$

because only one subscript value ($s_2 = 2$) has been defined;

for $s_1 = 5$, $n_2 = 2$

because two values ($s_2 = 7$ and 8) have been defined with the same subscript $s_1 = 5$;

for $s_1 = 7$ and 8, $n_2 = 1$

because one value ($s_2 = 2$, $s_2 = 9$), has been defined with each of the subscripts $s_1 = 7$ and $s_1 = 8$;

for $s_1 = 1$ and $s_2 = 2$, $n_2 = 2$

because there are two values ($s_3 = 3$, $s_3 = 4$) with the same subscripts $s_1 = 1$ and $s_2 = 2$.

The dimensioned labels of the form label$(s_1,s_2,\ldots,s_j)$, where $j < n$, are defined by the assembler to have values equal to the number of different subscripts used in the next higher dimension specification.

Example:

```
000001                                      /.
000002                         000144       A(1,2,3)  EQU       100
000003                         000310       A(5,7,3)  EQU       200
000004                         000454       A(5,8,3)  EQU       300
000005                         000620       A(7,2,2)  EQU       400
000006                         000764       A(8,9,0)  EQU       500
000007                         001130       A(1,2,4)  EQU       600
000008                                      .
000009          00 000000      000144                 +A(1,2,3)
000010          00 000001      000004                 +A
000011          00 000002      000001                 +A(1)
000012          00 000003      000002                 +A(5)
000013          00 000004      000001                 +A(7)
000014          00 000005      000002                 +A(1,2)
000015          00 000006      000000                 +A(5,9)
000016                                                END
```

Explanation:

- Lines 2 through 7 define a set of dimensioned labels A.
- Line 9 generates the value of the label A(1,2,3).
- Line 10 generates a number equal to the number of different subscript values s_1 defined in the set. A = 4 because $n_1 = 4$ (s_1 = 1,5,7,8).
- Line 11 generates a number equal to the number of different subscript values s_2 defined in the set A with $s_1 = 1$. A(1) = 1 because only $s_2 = 2$ has been defined with $s_1 = 1$.
- Line 12 generates the value n_2 for $s_1 = 5$.
  $n_2 = 2$ because s_2 = 7 and 8 for $s_1 = 5$.
- Line 13 generates the value n_2 for $s_1 = 7$.
  $n_2 = 1$ because only $s_2 = 2$ for $s_1 = 7$.
- Line 14 generates the value n_3 for $s_1 = 1$ and $s_2 = 2$.
  $n_3 = 2$ because s_3 = 3 and 4 for $s_1 = 1$, $s_2 = 2$.
- Line 15 generates the value n_3 for $s_1 = 5$ and $s_2 = 9$.
  $n_3 = 0$ because no value with $s_2 = 9$ has been defined.

Label	Value	Definition
A(1,2,3)	100	explicit*
A(5,7,3)	200	explicit*
A(5,8,3)	300	explicit*
A(7,2,2)	400	explicit*
A(8,9,0)	500	explicit*
A(1,2,4)	600	explicit*
A	4	implicit
A(1)	1	implicit
A(5)	2	implicit
A(7)	1	implicit
A(8)	1	implicit
A(1,2)	2	implicit
A(5,7)	1	implicit
A(5,8)	1	implicit
A(7,2)	1	implicit
A(8,9)	1	implicit
A(9) and all others	0	implicit

*See foregoing example.

The purpose of using dimensioned labels as opposed to simple labels may vary. The DO directive and procedures are capable of generating more than one word of data or series of instructions. Combined with these tools, dimensioned labels provide an extremely convenient method for manipulating arrays of any desired dimension.

1.3.1.4. Location Counter Declaration

When a program element is assembled, relocatable object code is produced as a result of the assembly. When the assembled program is loaded by the loader, the actual address values are assigned. The relocatable code produced by the assembler is therefore relative to a base address assigned by the loader when the program is executed. A location counter specifies under which base address a particular word is to be generated. There are 16 location counters (0–15) within any one assembly. Any location counter may be used or referenced in any sequence. The loader regroups the data generated under the various location counters so that each appears in memory as though the code within the location counter was generated contiguously.

A program remains under control of location counter 0 if no location counter is explicitly specified. When a specific location counter is specified, all subsequent coding is generated under its control until another location counter is specified.

→ A specific location counter may be activated by $(n) as the first entry in the label field, where n represents an expression whose value is within the range of 0 through 15 and denotes the location counter to be activated.

Coding may be present in the same statement which defines a new location counter. If this is done, the code generated will be under control of the new location counter. If a label is desired on a line of code which also defines a new location counter, the format is:

```
    $(n),label     operation     operand
```

If a symbol is used in defining the location counter, it must have been previously defined.

Example:

```
000001                                          /,
000002          00 000000   70 0005                             LLK      5
000003          00 000001   55 0003                             JI       (LABEL)
000004          05 000000   71 0003             $(5),LABEL      ALK      3
000005                                                          LIT
000006          05 000001   55 0002                             JI       ($(0))
000007                                          $(1-1),
000008          00 000002   770301                              ERRORS
000009                                                          END
                00 000003   000000
                05 000002   000002
```

Explanation:

- Line 2 generates an LLK 5 instruction under location counter 0.
- Line 3 transfers control to the address denoted by LABEL, which is not necessarily the next address because it is defined under a different location counter.
- Line 4 defines LABEL under location counter 5 and generates an ALK 3 instruction under location counter 5.
- Line 6 transfers control back to the next address under location counter 0.
- Line 7 reactivates location counter 0.
- Line 8 generates a procedure call ERROR$. The transfer made in line 6 will be to this address.

1.3.2. Operation Field

The operation field defines the purpose of the symbolic statement. The operation field starts with the first nonblank character following the label field. If no label field value is present, at least one blank character must be coded before defining the operation field. The operation field may contain any one of the following:

- a mnemonic operation code identifying which instruction is to be generated;
- an assembler directive specifying some special function to be performed by the assembler (see Section 2);
- a FORM reference specifying that a data word is to be constructed according to the format defined by the FORM directive (see Section 2);
- a procedure reference specifying that some procedure is to be assembled (see Section 3); or
- a data word generating code specifying that one or more words of data constants are to be generated.

The operation field must be terminated by at least one blank character unless:

- a procedure reference is made,
- a data generating code is defined, or
- a period is used to terminate the entire statement.

If a procedure reference is made, the operation field may be terminated by a comma followed by procedure parameters. If a data generation code is defined, the data word may immediately follow the identifier.

The content of the operation field determines the value of the active location counter. If an instruction is generated, the location counter is incremented by 1 or 2 depending on whether an 18- or 36-bit instruction is to be generated. If an assembler directive is referenced, the location counter value may or may not be advanced depending on the specific directive. A FORM reference may cause the location counter to be advanced by one or two depending on the specified FORM directive. A procedure reference may cause the location counter to be advanced by an indefinite value, depending entirely on the definition of the procedure sample.

A data word generating code will cause the location counter to be advanced depending on the number of words generated.

Example:

```
000001                                       /.
000002          00 000000    36 0003                   LBK       3
000003                                       FR        FORM      6,12
000004          00 000001    00 0002                   FR        0,LABEL
000005     U                                           PRCAL,PAR1
000006          00 000002    204511          LABEL     +10.3
                00 000003    463146
000007                                                  END
```

Explanation:

- Line 2 specifies generation of an LBK instruction.
- Line 3 is an assembler directive defining the format FR.
- Line 4 is a FORM reference.
- Line 5 is a procedure reference on the procedure PRCALL.
- Line 6 is a data constant.

1.3.3. Operand Field

The operand field starts with the first nonblank character following the operation field. The components of the operand field are called expressions or subfields and define the information necessary to complete the type of statement specified by the operation field.

The operand of a mnemonic instruction or data constant requires only one expression which is terminated by a blank character.

Several of the assembler directives do not require an operand. Others require several expressions. When groups of expressions are used, they are separated by commas. A group of such expressions is referred to as a list of expressions. Procedures may permit multiple lists of expressions. When omitting a subfield other than the first or last subfield, the construction comma–zero–comma (,0,) or two contiguous commas (,,) is necessary. Ending subfields may be omitted entirely if unnecessary.

Example:

LABEL (1)	OPERATION (10)	OPERAND (20, 30, 40)
	LLK	5
	SL	TAG
	MOVE	10 FROM HERE TO THERE
	CPL	.
IF	FORM	3,5,10

Explanation:

- Line 1 is a mnemonic instruction. The operand field contains an expression whose value is 5.
- Line 2 is a mnemonic instruction. The operand field contains an expression whose value is the relocatable address TAG.
- Line 3 is a procedure call containing five lists of expressions.
- Line 4 is a mnemonic instruction not requiring an operand.
- Line 5 is a FORM directive. The operand field contains one list of three expressions or subfields.

1.3.4. Comment Field

The construction space–period–space (b̄.b̄) terminates a line of coding. Any additional subfields implied by the operation field are taken to be zero. Any characters following the space–period–space are printed on the assembly listing and may be used as comments to clarify the purpose of the line of code. If the operand field has been totally specified, comments may immediately follow the blank character which terminates the operand field.

1.3.5. Line Continuation

A symbolic line may be continued to the next card image. When a semicolon is encountered during the processing of the label field, the operation field, or the operand field, the next card image is read and processing continues starting with the next nonblank character. If a new list is to be defined on a continuation card, at least one space should occur before the semicolon.

If a semicolon occurs in the comment field, whether defined or implied, it is not treated as a continuation character, and the next card image is processed separately. Continuation to the next card may be specified in any of the three basic fields. In some situations, such as the first reference to a library procedure, the label and operation field must be specified on the same card image. In general, it is recommended that semicolons only be used in the operand field.

Example:

```
000001                                          /.
000002          00 000000   000000              LABEL     +$
000003          00 000001                       LABELT    RES       10
000004                      000002              A         EQU       2
000005                      000003              B         EQU       3
000006    D                 777776              TAG       EQU       LABEL+1-(A>0)-(B<5)
000007                                          TAG       EQU       LAB;      COMMENTS MAY
000008                                                              ELT;           FOLLOW THE ;
000009                                                              +1-(A>0)-;
000010    D                 000000                                  (B<5)
000011                                                    END
```

1.3.6. Ejection of Paper

A slash (/) appearing in column 1 advances paper in the printer to the top of the next page. This line may not contain any coding but may contain comments. The slash prints on the new page (see 2.11).

1.4. EXPRESSIONS

An expression is an elementary item or a series of elementary items connected by operators. Blanks are not permitted within expressions. The values of elementary items can be combined through operators (see 1.4.2). The resulting value becomes the value of the expression. In addition to having an arithmetic value, each elementary item has associated with it a *mode value* which indicates whether the numeric value of the item is constant, that is, cannot be changed, or is relocatable, that is, relative to some base constant to be determined at some later time. This base constant is generally a storage address or drum address determined by the job loader prior to execution of the program. In combining elementary items to form an expression, the mode values of the items are also operated upon to form the mode value of the expression. When combining elementary items to form an expression, some care must be exercised to ensure that the resulting mode value of the expression is also correct (see 1.4.4).

In combining elementary items to form an expression, the symbolic statement is scanned and interpreted from left to right. Parentheses may be used to force items to be combined in a different order. All expressions within parentheses are evaluated before their results are available to be operated upon. Up to six nested levels of parentheses may be used.

1.4.1. Elementary Items

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

The magnitude of the value of an elementary item may not exceed 2^{36}-1, that is, 0777777777777. If an elementary item is not defined, it is assigned a value of zero. Expressions containing undefined (externally referenced) elementary items may not exceed a magnitude of 2^{18}-1, that is, 0777777.

There are eight ways in which elementary items may be represented. They are discussed in the following paragraphs.

1.4.1.1. Symbolic Label

Any label may be used as an elementary item. The value of the item is the relocatable location counter value of the statement associated with the label. If the label was defined with an EQU directive, the item value is that of the operand expression of the EQU statement. Undefined labels have a constant zero value.

Example:

```
000001                                        /.
000002        00 000000   12 0001             TAG2     LL        TAG
000003        00 000001   12 0000             TAG      LL        TAG2
000004                                                 END
```

Explanation:

- Line 2 defines TAG2 to have a value equal to the relocatable location counter value of the word containing the instruction LL TAG. The operand field contains an expression formed by a single elementary item TAG. The value of TAG is defined in line 3 as the relocatable location counter value of the word containing the instruction LL TAG2.

1.4.1.2. Location Counter

The relocatable value of any of the location counters may be used as an elementary item. The symbolic representation of a location counter value reference has the form:

$(expression)
or
$

If a dollar sign alone is used, the value of the elementary item is the current value of the active location counter. If a dollar sign followed by a left parenthesis is used, the expression value contained within the parenthesis defines which location counter is referenced. The value of the expression must be between 0 and 15. It should be remembered in using the $+n that some instructions increment the location counter value by 2.

Example:

```
000001                                             /.
000002      00 000000                                          RES     8
000003      02 000000   000010                     $(2)        +       $(0)
000004      00 000010   55 0001                    $(0)        JI      $(2)
000005      00 000011   34 0014                                J       $+3
000006      00 000012   502000                                 LSD     $(2)
            00 000013   000001
000007                                                         END
```

1.4.1.3. Octal Numbers

An octal number is an elementary item. An octal number consists of a group of octal integers (0–7) preceded by a 0. The value of the number is the value of the elementary item.

Example:

```
000001                                             /.
000002      00 000000   000077                                 +077
000003      00 000001   000301                                 +0301013013
            00 000002   013013
000004      00 000003   36 0017                                LBK     017
000005                                                         END
```

1.4.1.3.1. Double-Precision Octal Numbers

A double-precision octal value is produced by writing an octal constant larger than 18 bits or placing a letter D immediately after the last octal digit.

Example:

```
000001                                             /.
000002      00 000000   000000                                 +077D
            00 000001   000077
000003      00 000002   000001                                 +01000000
            00 000003   000000
000004      00 000004   000000                                 +1D+017
            00 000005   000020
000005                                                         END
```

1.4.1.4. Decimal Numbers

A decimal number is an elementary item. A decimal number consists of a group of decimal integers (0–9) the first of which is not a zero. The value of the elementary item is the value of the number.

Example:

```
000001                                                 /.
000002          00 000000   000115                           +77
000003          00 000001   000100                           +64
000004          00 000002   36 0017                          LBK      15
000005                                                       END
```

1.4.1.4.1. Double-Precision Decimal Numbers

A double-precision decimal value is produced by writing a decimal constant whose value is larger than 0777777 or by placing a letter D immediately after the last decimal digit.

Example:

```
000001                                                 /.
000002          00 000000   000000                           +77D
                00 000001   000115
000003          00 000002   000000                           +64D
                00 000003   000100
000004          00 000004   000000                           +1D+17
                00 000005   000022
000005                                                       END
```

1.4.1.5. Alpha Constants

Alphabetic, numeric, and special characters may be represented in 6-bit XS-3 code. When such characters are enclosed within apostrophes, the enclosed characters together form an alpha constant. The value associated with each character of the alpha constant is the 6-bit XS-3 code as defined in Table 1–1. The value of the elementary item is formed by stringing together the values associated with each character.

NOTE: A semicolon is a special character which is generated when enclosed by apostrophes. Therefore, it may not be used as a continuation character in an alpha constant or alpha string.

Example:

```
000001                                                 /.
000002          00 000000   000024                           +'A'
000003          00 000001   002425                           +'AB'
000004          00 000002   242526                           +'ABC'
000005          00 000003   70 0024                          LLK      'A'
000006          00 000004   71 0001                          ALK      'B'-'A'
000007                                                       END
```

The 6-bit value associated with a character in an alpha constant may be redefined through the use of the CHAR assembler directive (see 2.7).

An apostrophe may be present as a character within the alpha constant by coding two contiguous apostrophes for each apostrophe in the constant.

Example:

```
000001                                            /.
000002         00 000000    000056                             +''''
000003         00 000001    565624                             +'''''A'
000004                                                         END
```

If the alpha constant consists of one, two, or three characters, the value of the elementary item is right-justified, zerofilled. If the alpha constant consists of four, five, or six characters, the value of the elementary item is left-justified, spacefilled, and generates two words.

An alpha constant may not consist of more than six characters (see 1.5.2).

Example:

```
000001                                            /.
000002         00 000000    000024                             +'A'
000003         00 000001    002425                             +'AB'
000004         00 000002    242526                             +'ABC'
000005         00 000003    242526                             +'ABCD'
               00 000004    270000
000006         00 000005    242526                             +'ABC '
               00 000006    000000
000007                                                         END
```

1.4.1.5.1. Double-Precision Alpha Constants

A double-precision alpha constant is one which consists of four, five, or six characters, or one which is immediately followed by the letter D.

Example:

```
000001                                            /.
000002         00 000000    000000                             +'A'D
               00 000001    000024
000003         00 000002    242526                             +'ABC DE'
               00 000003    002730
000004                                                         END
```

1.4.1.6. Floating-Point Numbers

A floating-point number is an elementary item. The value of the elementary item is the 36-bit binary number formatted according to the hardware representation of floating-point numbers. Note that in manipulating floating-point elementary items, the assembler uses double-precision *integer* arithmetic so that expressions of the type

1.0 + 1

result in a binary number which is the result of an integer arithmetic addition of the two elementary items.

A floating-point number is recognized by the presence of a decimal point immediately following a decimal number. The format of a floating point number is one of the following:

d.
d.d
d.dEse
d.Ese
d.Ee

where: d represents one or more decimal digits.
s represents the sign of the characteristic and may be either + or –.
e represents one or more decimal digits which define the power of 10 by which the number is to be multiplied.

Example:

```
000001                                                   /.
000002            00 000000   201400                              +1.
                  00 000001   000000
000003            00 000002   201403                              +1.015
                  00 000003   656050
000004            00 000004   200400                              +0.5
                  00 000005   000000
000005            00 000006   203500                              +0.5E+1
                  00 000007   000000
000006            00 000010   174631                              +0.5E-1
                  00 000011   463146
000007            00 000012   216471                              +100.324E2
                  00 000013   406314
000008                                                            END
```

1.4.1.7. Parameter Reference Form

The parameter reference form (PARAFORM) is an elementary item as long as the procedure sample is being processed. The definition, explanation, and use of paraforms are given in Section 3.

1.4.1.8. Line Items (Literals)

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

A literal is represented as an expression enclosed within parentheses and without connecting operators. The assembler then generates a word containing the expression value, and this word appears in a literal table 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 (see 2.16).

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

Line items within line items are permitted up to five levels. If an operator immediately precedes an item enclosed within parentheses, the item is not a literal.

Example:

```
000001                                             /*
000002        00 000000   12 0007                            LL        ('END')
000003        00 000001   32 0010                            LB        (0101)
000004        00 000002   10 0011                            LU        (J   5+5)
000005        00 000003   10 0013                            LU        (LU   (017))
000006        00 000004   70 4600                            LLK       +(SLL    0)
000007        00 000005   10 0014                            LA        (1,0)
              00 000006   12 0015
000008                                                       END
              00 000007   305027
              00 000010   000101
              00 000011   34 0007
              00 000012   000017
              00 000013   10 0012
              00 000014   201400
              00 000015   000000
```

1.4.2. Operators

There are 12 operators in the assembler which designate the method, and implicitly the sequence, to be employed in combining elementary items within a subfield. Blanks are not permitted within an expression. Evaluation of an expression begins with the substitution of values for each elementary item. The operations are then performed from left to right in hierarchical order as listed in Table 1–2. All the operators listed are assembly-time operators.

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 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.

HIERARCHY	OPERATOR	DESCRIPTION
Highest 6	*/	a*/b is equivalent to $a*2^b$
5	* / //	arithmetic product arithmetic quotient covered quotient (a//b is equivalent to $\frac{a+b-1}{b}$)
4	+ -	arithmetic sum arithmetic difference
3	**	logical product (AND) $\begin{array}{c\|cc} & 1 & 0 \\ \hline 1 & 1 & 0 \\ 0 & 0 & 0 \end{array}$
2	++	logical sum (OR) $\begin{array}{c\|cc} & 1 & 0 \\ \hline 1 & 1 & 1 \\ 0 & 1 & 0 \end{array}$
2	--	logical difference (EXCLUSIVE OR) $\begin{array}{c\|cc} & 1 & 0 \\ \hline 1 & 0 & 1 \\ 0 & 1 & 0 \end{array}$
Lowest 1	= > <	a = b has the value of 1 if true, 0 if otherwise a > b has the value of 1 if true, 0 if otherwise a < b has the value of 1 if true, 0 if otherwise

Table 1–2. Hierarchy of Operators

In the absence of parentheses, the rules of priority determine the sequence in which operations are performed within an expression. When two or more operators of the same priority are used, the sequence of interpretation is from left to right. The following two sample problems illustrate this point:

PROBLEM 1:	9-2*3++12**6	The result is 7.
after step 1	9-6++12**6	
after step 2	3++12**6	
after step 3	3++4	
after step 4	7	

PROBLEM 2:	((9-(2*3/4))++12)**6	The result is 4.
after step 1	((9-1)++12)**6	
after step 2	(8++12)**6	
after step 3	12**6	
after step 4	4	

1.4.2.1. 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 produces a right shift). x*/b is equivalent to $x*2^b$.

If the sign of the exponent is positive, a left-circular shift of the number is performed. If the sign of the exponent is negative, a right-arithmetic shift of the number is performed.

Example:

```
000001
000002          00 000000   000060                          +6*/3          .=6*8
000003          00 000001   000003                          +6*/-1         .=6/2
000004          00 000002   777770                          -073*/-3       .=-073/8
000005          00 000003   700000                          +(0777777*/18)*/-3
000006                                                      END
```

1.4.2.2. Arithmetic Product (*)

The integer value of the first item, the multiplicand, is multiplied by the integer value of the second item, the multiplier, to produce a product which becomes the value of the expression or next item.

Example:

```
                                                    /.
000001                                                      +4*4
000002          00 000000   000020                          +4*1*/3        .=4*8
000003          00 000001   000040                          +(4*2)*/3      .=8*8
000004          00 000002   000100                  IL      EQU        5
000005                      000005                  BF      EQU        2
000006                      000002                  BL      EQU        IL*BF
000007                      000012                          END
000008
```

1.4.2.3. Arithmetic Quotient (/)

The integer value of the first item, the dividend, is divided by the integer value of the second element, the divisor, and the resultant quotient becomes the value of the expression or next item.

Example:

```
000001                                                  /*
000002          00 000000    000002                                +4/2
000003          00 000001    000002                                +4*2/3
000004          00 000002    000000                                +4*(2/3)
000005          00 000003    000000                                +4*2/3*/3      *=(4*2)/24
000006                                                             END
```

Note that the remainder of the division is discarded and that the quotient resulting from a divide must be less than 2^{18}-1.

1.4.2.4. Covered Quotient (//)

The covered quotient operates the same way as the arithmetic quotient with the following exception. If the remainder of the division is greater than zero, one is added to the integer value of the quotient. The resulting integer is substituted in the expression. The covered quotient may be expressed in the following formula:

$$a//b = \frac{a + b - 1}{b}$$

Example:

```
000001
000002          00 000000    000002                     /*
000003          00 000001    000004                                +5//3
000004          00 000002    000004                                +2*5//3
000005                                                             +2*(5//3)
                                                                   END
```

1.4.2.5. Arithmetic Sum (+)

The arithmetic sum operator produces the algebraic integer sum of the values of two items.

Example:

```
000001                                                  /*
000002          00 000000    000007                                +5+2
000003          00 000001    12 0003                               LL     $+2
000004          00 000002    000065                                +5+2*3*/3
000005          00 000003    000250                                +(5+2)*3*/3
000006          00 000004    000250                                +((5+2)*3)*/3
000007                                                             END
```

1.4.2.6. Arithmetic Difference (-)

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

Example:

```
000001                                                              /•
000002              00 000000    12 0002                                          LL      $+4-2
000003              00 000001    000003                                           +5-2
000004              00 000002    777776                                           +5-2*3
000005              00 000003    000011                                           +(5-2)*3
000006                                                                            END
```

1.4.2.7. Logical Product (**)

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

Example:

```
      03**05   The result is 01.
      000011
   ** 000101
   ---------
      000001
```

1.4.2.8. Logical Sum (++)

The logical sum operator (OR) produces the logical sum of two items.

Example:

```
      03++05   The result is 07.
      000011
   ++ 000101
   ---------
      000111
```

1.4.2.9. Logical Difference (--)

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

Example:

```
      03--05   The result is 06.
      000011
   -- 000101
   ---------
      000110
```

1.4.2.10. Equal (=)

The integer value of the first item is compared with the integer value of the second item. If the two values are equal, the result of the operation is a binary 1. If they are not equal, the result of the operation is a binary 0.

Example:

```
000001                                           /*
000002                        000005             A           EQU       5
000003         00 000000      70 0003                        LLK       (A=5)*3+(A=6)*2
000004                                                       END
```

1.4.2.11. Greater Than (>)

The integer value of the first item is compared with the integer value of the second item. If the value of the first item is greater than the value of the second, the result of the operation is a binary 1. If the integer value of the first item is less than or equal to the second, the result of the operation is a binary 0.

Example:

```
000001                                           /*
000002                        000005             A           EQU       5
000003         00 000000      36 0000                        LBK       (A>5)*3+(A<5)*2
000004         00 000001      70 0000                        LLK       (A>15)*3
000005         00 000002      71 0003                        ALK       (A>2)*3
000006                                                       END
```

1.4.2.12. Less Than (<)

The integer value of the first item is compared with the integer value of the second item. If the value of the first item is less than the value of the second item, the result of the operation is a binary 1. If the first value is greater than or equal to the value of the second, the result of the operation is a binary 0.

Example:

```
000001                                           /*
000002                        000005             A           EQU       5
000003         00 000000      36 0000                        LBK       3*(A<5)
000004         00 000001      70 0003                        LLK       3*(A<15)
000005         00 000002      71 0000                        ALK       3*(A<2)
000006                                                       END
```

1.4.3. Interbay Offset Operator (!)

The interbay offset operator (IBOO) is a special operator recognized by the assembler which operates only on the mode of an expression. When the IBOO operator is present in an expression, a flag is set in the relocation output which causes the loader to relocate the data word in a special manner. If used, the IBOO operator must follow an elementary item, and may be followed by an operator.

Example:

```
000001                                   /•
000002        00 000000   32 0003                  LB      (LABEL!)
000003        00 000001   32 0004                  LB      (LABEL!+1)
000004        00 000002   000000        LABEL      +0
000005                                             END
              00 000003   000002
              00 000004   000003
```

The purpose of the IBOO operator is to facilitate the accessing of storage in different bays.

Consider the following ways of accessing the contents of location FROM, which may be located anywhere in storage.

Examples:

```
000001                                   /•
000002        00 000000   12 0012                  LL      (S)
000003        00 000001   52 0013                  AND     (0770000)
000004        00 000002   44 0010                  SL      BAY
000005                                   •
000006    U   00 000003   12 0014                  LL      (FROM)
000007        00 000004   16 0010                  ANL     BAY
000008        00 000005   44 0011                  SL      FROMR
000009                                   •
000010        00 000006   32 0011                  LB      FROMR
000011        00 000007   13 0000                  LL      •0      . (AL)=(FROM)
000012        00 000010   000000        BAY        +0
000013        00 000011   000000        FROMR      +0
000014                                             END
              0C 000012   000000
              00 000013   770000
          U   00 000014   000000

000001                                   /•
000002    U   00 000000   32 0004                  LB      (FROM)
000003        00 000001   5073 20                  LSR     020
000004        00 000002   13 0000                  LL      •0
000005        00 000003   5073 00                  LSR     0       . (AL)=(FROM)
000006                                             END
          U   00 000004   000000

000001                                   /•
000002    U   00 000000   5073 20                  LSR     020+FROM
000003    U   00 000001   12 0000                  LL      FROM
000004        00 000002   5073 00                  LSR     0       . (AL)=(FROM)
000005                                             END
```

Each of the three foregoing methods has particular advantages. The first example uses six instructions to set up a bay-relative address. Subsequent references use two instructions. The disadvantage comes about if many different locations are to be accessed in this manner.

The second example is disadvantageous if frequent accesses have to be made because four instructions are used each time.

The third example still uses three instructions each time and is valid only if FROM is an external reference. If FROM is defined within the assembled program, the LSR operand specification should be coded as:

LSR 020 + FROM - (FROM**0777777).

The IBOO operator causes the loader to relocate the specified value as follows:

(VALUE)+(REL. BASE)-(BAY IN WHICH VALUE IS STORED) .

As a result, the above access may be performed as follows:

```
000001                                             /.
000002    U        00 000000   32 0002                     LB        (FROM;
000003             00 000001   13 0000                     LL        .0
000004                                                     END
          U        00 000002   000000
```

1.4.4. Expression Modes

As stated previously, each elementary item has both an arithmetic and a mode value. When operators are used to combine elementary items to form an expression, the mode values of the elementary items are combined also to form the mode of the expression.

Table 1–3 gives the rules for determining whether the result of a binary operation is relocatable.

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

NOTES:

1. The difference between two relocatable quantities under the same location counter is not relocatable.
2. Except as noted for level 4, the relocation error flag (R) is set for these operations.
3. 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.

Table 1–3. Rules for Determining whether Results of Binary Operations are Relocatable

The mode values associated with a line of code may be examined by using the M option on the ASM control card (see Section 4).

1.5. DATA WORD GENERATION

A + or – in the operation field followed by a single subfield generates one or more data words. The + or – sign may be separated from the subfield by any number of blanks. If the first item in the expression is a number or an alpha constant, the + or – may be omitted. If the mode value of the operand expression signifies that the data word is double-precision, two 18-bit words are generated. In the absence of a + sign, the value of a number is taken to be positive.

The operand field of a data generation statement may contain:

- an expression or elementary item
- an alpha string
- a double-precision floating-point number

1.5.1. Data Word Expressions

The operand field or operation field may contain an expression. A data word consisting of the value of the expression is generated.

Example:

```
000001                                          /.
000002      00 000000   000005                             +5
000003      00 000001   000002                  TAG        +$+1
000004      00 000002   000004                             +TAG+3
000005      00 000003   201400                             +1,0
            00 000004   000000
000006      00 000005   000000                             +5D
            00 000006   000005
000007      00 000007   000001                             +(TAG)
000008      00 000010   000012                             (TAG)
000009      00 000011   001137                             +20*(27+2)++037
000010                                                     END
            00 000012   000001
```

1.5.2. Alpha Strings

An alpha string consists of a series of alphabetic, numeric, and special characters enclosed within apostrophes. Two successive apostrophes within the string are equivalent to a single apostrophe which does not signify the end of the string. For each three characters in the string, one 18-bit data word is generated which consists of an alpha constant equal to the binary equivalent of the three characters.

Characters are left-justified, spacefilled unless the string consists of less than three characters. In this case, an alpha constant (right-justified, zerofilled) is generated.

Example:

```
000001      00 000000   663334                             +'THIS IS AN ALPHA STRING'
            00 000001   650034
            00 000002   650024
            00 000003   500024
            00 000004   465233
            00 000005   240065
            00 000006   665434
            00 000007   503200
000002      00 000010   000024                             +'A'
000003                                                     END
```

1.5.3. Double-Precision Floating-Point Numbers

Double-precision floating-point numbers may be generated which conform in format to the conventions established in the FORTRAN compiler. A double-precision floating-point number consists of three 18-bit words. The first word contains the characteristic; the second and third contain the mantissa. If a floating-point elementary item occurs which specifies more than 27 bits of significance, or which contains the letter D in the exponent instead of the letter E, a double-precision floating-point format is generated.

Example:

```
000001
000002        00 000000   040007                     +1,D2
              00 000001   310000
              00 000002   000000
000003        00 000003   040001                     +1,23456789
              00 000004   236014
              00 000005   510210
000004        00 000006   037755                     -0,12D-5
              00 000007   536740
              00 000010   501437
000005                                               END
```

1.6. DOUBLE-PRECISION EXPRESSIONS

As previously stated, several elementary items may be specified to be double-precision. If an expression contains a double-precision item, the expression is said to be a double-precision expression. When a double-precision expression is used to generate data, two words are generated. If the line item specified in a literal is a double-precision item, the literal value is the address of the first of the two words generated in the literal table.

The following restrictions exist when generating double-precision data words.

- An expression which contains an external reference may not be defined as a double-precision expression.
- Simple labels may be defined to have a value which exceeds $2^{18}-1$, but if such labels are used to generate a data constant, only one word is generated which consists of the least significant 18 bits of the value of the label.

Example:

```
000001                                   /.
000002                    000000         A        EQU      01000000
000003        00 000000   000000                  +A
000004                                   .
000005                    000000         D(1)     EQU      A
000006        00 000001   000000                  +D(1)
000007        00 000002   10 0010                 LA       ('ABCDEF')
              00 000003   12 0011
000008        00 000004   10 0012        TAG      LA       (1.0)
              00 000005   12 0013
000009        00 000006   10 0014                 LA       (RS    TAG)
              00 000007   12 0015
000010                                            END
              00 000010   242526
              00 000011   273031
              00 000012   201400
              00 000013   000000
              00 000014   5010 00
              00 000015   00 0004
```

2. ASSEMBLER DIRECTIVES

2.1. GENERAL

The assembler provides a series of special directives which provide the means to control or direct the generation of object code. The symbolic assembler directives control or direct the assembly processor just as the hardware operation codes control or direct the central processor. The assembler directives are represented by mnemonics written in the operation field of a symbolic line of code. The directives are used to equate the expressions, control the location counter, format the object code, and control the generation of object code. The general format for the directives is:

```
label        directive        specification
```

The manner in which the assembler interprets each directive varies and is described in detail in this section.

2.2. EQU DIRECTIVE

The EQU directive is used to equate the symbolic label in the label field to the value of the expression in the operand field. Thereafter, this label may be used or referenced in operand expressions. The operand consists of one list of one expression. The format is:

```
label        EQU        e
```

Except in the case of dimensioned labels, redefinition of a label causes the statement to be flagged as duplicate; however, the value of the latest expression is used when a reference to the symbolic label is made. All statements referencing such a label are also flagged. When a directive is written which affects the value of the location counter and which uses a label defined in an EQU directive to do this, the EQU directive which defines the value of the label must occur first.

When the operand expression of the EQU directive is another label, this label must have been previously defined in the program assembly or not defined at all. If the label referenced is defined after it is referenced, the statement is flagged as doubly defined. If the label referenced is not defined, it becomes an external reference. Subsequent references to the label defined through an EQU directive as equal to an external label reference the external label. The label defined in this manner may not itself be externally defined.

Example:

```
00:10:51          @ASM,M      T2-1

                  UNIVAC   418-III ASSEMBLY --    MAR 11 1970     00:10:51
000001                                                  /.
000002                             000001               CODE      EQU       1
000003                             000100               XCDE      EQU       64*(CODE=1)
000004                             770101               ZCDE      EQU       0770001+XCDE
000005              00 000000      70 0005              LABEL     LLK       5
                                     + 6,12 00
000006                             000000               LAB2      EQU       LABEL
000007                                                  .
000008    D                        000005               DLB       EQU       5
000009    D                        000006               DLB       EQU       6
000010    D         00 000001      000006                         +DLB
                                     +    18 00
000011                                                  .
000012    U         00 000002      000000                         +DLB2
                                     +    18 00    CODE
000013    D                        000001               DLB2      EQU       DLB3
000014    D         00 000003      000001                         +DLB2
                                     +    18 00
000015                             000001               DLB3      EQU       1
000016    D         00 000004      000001                         +DLB2
                                     +    18 00
000017                                                  .
000018    U                        000000               ULB       EQU       EXDEF
000019    U         00 000005      70 0001              LLK                 ULB+1
                                     + 6,12 00     EXDEF
000020                                                  .
000021    U                        000000               ELB*      EQU       EXDEF          .  ILLEGAL
000022                                                            END
                                                              *** SUMMARY ***

     PROGRAM SIZE:   00 00006

     EXTERNAL OR UNDEFINED REFERENCES:    EXDEF

     EXTERNAL DEFINITIONS:                ELB

     DOUBLY DEFINED LABELS:               DLB2      DLB
```

Explanation:

- Line 2 defines CODE to have a value of 1.
- Line 3 defines XCDE to have a value of 64.
- Line 4 defines ZCDE to have a value of 0770101.
- Line 6 defines LAB2 to have a value which is relocatable and equal to the location counter value assigned to line 5.
- Lines 8 through 10 illustrate that D flags are generated if a label is redefined.
- Lines 12 through 15 illustrate forward referencing of a label and the associated dangers in that a reference to the label is different depending on where the reference is made.

- Lines 17 and 18 illustrate indirect external referencing.
- Line 20 illustrates an illegal use of external referencing and external definition.

The magnitude of the value of the operand field may be 36 bits. However, double-word data generation may only be used through EQU directives using dimensioned labels.

Example:

```
                                                  /*
000001                                            A        EQU      01000000
000002                          000000                     *A
000003         00 000000        000000            B(1)     EQU      01000000
000004                          000000                     *B(1)
000005         00 000001        000001
               00 000002        000000                     END
000006
```

Explanation:

- Line 2 defines the label A to have a value of 01000000.
- Line 3 generates only one data word equal to the least significant 18 bits of the value of A (sign extended). A zero is therefore generated.
- Line 4 defines the value of B(1) to be 01000000.
- Line 5 generates two data words, 1 and 0.

2.3. RES DIRECTIVE

The RES directive is used to redefine the value of the active location counter. If the sign of the expression in the operand field is positive, an area of main storage is reserved (buffer). The label, if used, is assigned to the location counter value prior to changing it; that is, it refers to the first word of the reserved area if the operand field is positive. The format is:

label   RES   e

Symbols appearing in the operand field must be defined prior to the use of the RES directive.

In redefining the value of the location counter, no code is generated; that is, zeros are not generated for the reserved area. Because the loading of a program is preceded by clearing its main storage area, the RES directive, when used to define work area buffers, effectively defines their value as zero.

Example:

```
000001                                            /*
000002                          000002            I        EQU      2
000003         00 000000                          WORK     RES      56
000004         00 000070        70 0005                    LLK      5
000005         00 000071                                   DO   I>1 , RES -1
000006         00 000070        70 0003                    LLK      3
000007                                                     END
```

Explanation:

- Line 3 reserves a 56-word work area.
- Line 4 generates an instruction LLK 5.
- Line 5 generates an LLK 3 at the location following the LLK 5; or, if I>1, the LLK 3 in line 6 is generated at the same location counter value resulting in erasing the LLK 5.

NOTE: The relocatable object code produced by the assembler is such that the generated code is read in a single drum access by the loader as long as the code is continuous, that is, as long as the location counter value increases continuously. The RES directive may cause a break in the sequence of code generated. In the same way, a change in the location counter under which code is generated causes a break in the sequence. As a result, the program load time is increased because multiple drum accesses have to be made. Judicious use of the RES directive results in faster loading of the relocatable code.

Example:

LABEL	OPERATION	OPERAND
	RES	2
	+0	
	+0	

Explanation:

- Line 1 changes the location counter value by 2. As a result, a different drum access is made by the loader.

2.4. FORM DIRECTIVE

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 18 or 36 depending on whether a single or double form word is desired. If the sum of the values of the operand expressions does not equal 18 or 36, an expression error results.

By writing the label of the FORM directive in the operation field, 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.

Unless the field size of the last expression is 12 or 17 bits, the data word generated is a constant. If the last expression has a field size of 12 or 17 bits, the data word generated may be 12- or 17-bit relocatable, depending on the mode of the last parameter supplied on the FORM call line.

Example:

```
000001                                    /•
000002                                    PTGF      FORM    12,2,4
000003                      007706        P         EQU     07706
000004      00 000000       770662                  PTGF    P,3,2
000005                                    IS        FORM    6,12
000006      00 000001       32 0015                 LB      (IS 0,BUFAD)
000007      00 000002       13 0000                 LL      •0
000008      00 000003                     BUFAD     RES     10
000009                                              END
            00 000015       00 0003
```

Explanation:

- Line 2 defines a form PTGF. Three fields are defined consisting of 12, 2, and 4 bits, respectively.
- Line 3 defines a constant P = 07706.
- Line 4 references the PTGF form and generates a data word 0770662. The first 12 bits are built from P, the next 2 bits contain a 3, and the last 4 bits contain a 2. (Note that this is an example of a PTG$ call line).
- Line 5 defines a form IS. Two fields are defined consisting of 6 and 12 bits, respectively.
- Line 6 generates an LB instruction. The literal is defined to consist of a FORM reference. The first 6 bits are zero; the last 12 bits contain the address BUFAD. Since BUFAD is relocatable, the literal becomes 12-bit relocatable.
- Line 7 generates the code to load AL with the contents of BUFAD.

2.5. ODD DIRECTIVE

The ODD directive sets the currently active location counter so that the next symbolic line is assembled at the next odd address. If the location counter is already positioned at an odd address, no action is taken. The format is:

ODD

2.6. EVEN DIRECTIVE

The EVEN directive sets the currently active location counter so that the next symbolic line of code is assembled at the next even address. If the location counter is already positioned at an even address, no action is taken. The format is:

EVEN

2.7. CHAR DIRECTIVE

The CHAR directive permits selective redefinition of the values associated with alpha constants or strings (see 1.4.1.5). Unless a CHAR directive is used, the assembler uses the XS-3 code values defined in Table 1–1. — Sec 1 P5
10100

The alphabetic character A, for example, has an XS-3 value of 024. By using the CHAR directive, A may be redefined to have the value 6 (Fieldata). Unless redefined by another CHAR directive, all subsequent alpha constants and strings use the value of 6 for an A. The format of the CHAR directive is:

CHAR $e_1,f_1,e_2,f_2,\ldots,e_n,f_n$

The specification field consists of a list of paired expressions e_i and f_i; e_i specifies which character is to be changed, and f_i specifies the value to which the character e_i is to be changed. In order to identify which character is to be changed, its XS-3 value is specified in e_i.

Example:

```
000001                                                    /.
000002                                                         CHAR     'A',6,'B',7,'C',8
000003          00 000000    060710                            'ABC'
000004                                                         CHAR     024,024,025,025,026,026
000005          00 000001    242526                            'ABC'
000006                                                         END
```

Explanation:

- Line 2 redefines an 'A' (value 024) to 6, 'B' (value 025) to 7, and 'C' (value 026) to 8.
- Line 3 generates the alpha constant 'ABC'. As a result of the CHAR directive in line 1, the value 060710 is generated.
- Line 4 resets the values associated with 'A', 'B', and 'C' to 024, 025, and 026. Note that the characters to be changed must be referenced through their octal values because the alpha constants 'A', 'B', and 'C' have been redefined in line 2.
- Line 5 generates the alpha constant 'ABC'. Line 4 results in a value of 0242526.

2.7.1. XCHAR Directive

The XCHAR directive resets the values associated with alpha constants or strings to the XS-3 code values defined in Table 1–1. The format of the XCHAR directive is:

```
XCHAR
```

No label or operand field is present.

In the example in 2.7, the alpha constant value associated with 'A', 'B', and 'C' could have been redefined to their XS-3 value by using the XCHAR directive.

2.8. INSERT DIRECTIVE

The INSERT directive provides a method to insert symbolic code from either the user or the system library into the program which is currently being assembled. The operand consists of one list of one expression specifying the symbolic name of the program element to be inserted. The format is:

```
INSERT    e
```

Insertion of symbolic code is terminated when the end-of-file sentinel following the symbolic code in the library is detected. The symbolic element to be inserted may consist of common subroutines, translating routines, translation tables, and so on. The symbolic element to be inserted may itself have an INSERT directive.

2.9. UNLIST DIRECTIVE

The UNLIST directive provides a means of selectively preventing the printing of output of sections of a program. The format is:

```
UNLIST
```

2.10. LIST DIRECTIVE

The LIST directive provides a means of conditionally resuming printing of a program after using the UNLIST directive. The format is:

```
LIST    e
```

The LIST directive may have an operand. If the value of the operand expression is nonzero, printing resumes. If the value of the operand expression is zero, printing is discontinued.

2.11. SKIP DIRECTIVE

The SKIP directive provides a means of controlling page formatting of the assembly. The format is:

```
SKIP    e
```

The SKIP directive may have an operand expression e. If present, e lines are skipped before resuming the assembly print. If no operand field is specified, the paper is advanced to the next page before printing is resumed.

A page eject may also be accomplished by specifying a slash (/) in column 1 of any comment card.

Example:

```
     LABEL              OPERATION                   OPERAND
1             10                  20             30             40
/.
                  SKIP 4

```

2.12. END DIRECTIVE

The END directive is used to indicate that the last line of symbolic code in a procedure or in a program has been reached. The END directive may have an operand consisting of one list of two expressions. The operand is used to indicate the starting address and operating priority, respectively, of the assembled main program. A blank operand field indicates the end of a subroutine or procedure. The format is:

END e_1,e_2

When the END directive terminates a program assembly, all literals accumulated during the course of the assembly are listed and generated.

2.13. GO DIRECTIVE

The GO directive, when not used within a procedure, directs the assembler to ignore all statements until the associated NAME directive, not defined within a procedure, is encountered. The NAME directive must be defined subsequent to the GO directive (forward reference). If an END directive, not signifying the end of a procedure sample, is encountered before the NAME directive, the assembly is terminated as though the NAME directive immediately preceded the END directive. The format of the GO directive is:

GO label

where label represents the label of a NAME directive (see 3.12.2).

Example: See 2.14.

2.14. NAME DIRECTIVE

The NAME directive, when not defined within a procedure sample, is used to signify a point in the assembly at which assembly of symbolic statements is to be resumed after a GO directive. The format is:

label NAME

The label field contains a six-character label which may be referenced in the operand field of the GO directive (see 3.12.1).

Example:

```
000001                                    /.
000002                     000005         A          EQU       5
000003       00 000000     70 0003                   LLK       3
000004                                               DO   A=5 , GO NEXT
000005                                               LLK       5
000006                                    NEXT       NAME
000007                                               END
```

Explanation:

- Line 2 assigns a value of 5 to the label A.
- Line 4 uses a DO directive (see 2.15) which causes the statement GO NEXT to be performed.
- Line 5 is ignored during the assembly because of the GO statement in line 4.
- Line 6 defines the label NEXT. Assembly of source code resumes starting at the next statement.

2.15. DO DIRECTIVE

The DO directive is used to process a statement conditionally or to generate data tables by processing a single statement more than once. The format of a DO line is:

label1 DO expression ,label2 operation operand

The comma divides the DO line into two parts:

the determinant:	label1	DO	expression
the DO-item:	label2	operation	operand

The expression following the DO directive determines how many times the DO-item is performed. 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.

Example of a simple DO:

```
000001                                    /.
000002       00 000000     000001         A          DO 5 , +A
             00 000001     000002
             00 000002     000003
             00 000003     000004
             00 000004     000005
000003                                               END
```

Explanation:

- The DO-item generates a data word +A.
- The DO-item is performed 5 times.
- Each time the value of A is incremented by 1. The first time that the DO-item is performed, the value of A is 1.

All symbols appearing in the determinant expression must be defined prior to the DO statement. If undefined symbols appear, their value will be taken as 0. If the determinant expression has a negative value, it is reset to 0 and the DO-item is not performed.

2.15.1. Conditional DO

The operators <, =, and > are relational operators and generate expressions with a value of 0 (false) or 1 (true). Whenever the determinant expression of a DO statement has a value of 0 or 1, the DO is said to be conditional. If the determinant expression value is 0, the DO-item is not performed. If the determinant expression value is 1, the DO-item is performed.

Example:

```
 LABEL          OPERATION                    OPERAND
1         10              20              30              40
          DO  ($>07777) , LSR 020
```

Explanation:

- If the current location counter value is greater than 07777, the determinant expression has a value of 1. As a result, the LSR 020 instruction is generated.

2.15.2. Nesting of DO Directives

As stated previously, the DO-item may itself be the determinant of a second DO directive. DO statements may therefore be nested to as many levels as desired.

As the final DO-item is performed, the repeat count of the innermost determinant is satisfied before processing of the next determinant resumes.

Example:

```
000001                                                  /*
000002              00 000000   000011                  I          DO  3 ,J  DO  2 , *8*I+J
                    00 000001   000012
                    00 000002   000021
                    00 000003   000022
                    00 000004   000031
                    00 000005   000032
000003                                                             END
```

Explanation:

- The DO-item +8 *I+J is generated a total of 6 times. The value I is varied from 1 to 3. For each value of I, the DO-item is performed twice. The resultant data words generated are +011, +012, +021, +022, +031, +032.

2.16. LIT DIRECTIVE

The LIT directive is used to define a literal table under control of the active location counter. The format of a LIT statement is:

label LIT e

The label is optional and identifies the name of the literal table. The operand expression e is optional and determines the relative starting address of the literal table.

Through the use of the LIT directive, a number of separate literal tables can be created. Duplicate literals are eliminated within each unique literal table; however, duplicate literals may exist in separate literal tables. In the absence of a LIT directive, all literals are placed in the literal table under location counter zero. The entries in the label field of a LIT directive comply with the labeling rules as applied with the location counter declaration and label structure. However, the label may not be subscripted or suffixed by an asterisk nor may it be referenced (addresses or paraforms).

A LIT directive may have a label. If a label is present, the literal table is identified by this label. Literals generated under a labeled literal table have the form:

label(literal)

The label refers to the literal-table name, and literal represents the literal expression.

Example:

```
  LABEL              OPERATION                       OPERAND
1               10                  20                  30                  40
LBTAB1          LIT
                LL                  LBTAB1(  LLK  1)
```

If the label field of the LIT directive is left blank, literals to be placed in the defined table have the form:

(literal)

Example:

```
  LABEL              OPERATION                       OPERAND
1               10                  20                  30                  40
$(1)            LIT
                LL                  (  LLK  1)
```

Unless an operand field is present in the LIT statement, the literal table is generated under the location counter active at the time that the LIT directive occurred.

If an operand expression is present in the LIT statement, the literal table is generated starting at the address specified in the operand field of the LIT statement. The location counter of the specified starting address is used.

Literals are generated only in the second assembly pass. As a result, some care must be taken in defining the LIT directive. If the operand field specifies the literal table start address, only those literals subsequently defined for that literal table are assigned in the specified area.

Example:

```
000001                                         /.
000002      00 000000    12 0022                            LL       (1)
000003                                         TAB1         LIT
000004      00 000001    12 0023                            LL       TAB1(1)
000005                                         $(2),TAB2    LIT      AREA2
000006      00 000002    12 0010               $(0)         LL       TAB2(2)
000007      00 000003    10 0024                            LA       (1.0)
            00 000004    12 0025
000008                                         $(1)         LIT
000009      00 000005    12 0012               $(0)         LL       (7)
000010                                                      LIT      AREA1
000011      00 000006    12 0012                            LL       (7)
000012      00 000007    12 0000                            LL       (10)
000013      01 000000                          $(1),AREA1   RES      10
000014      00 000010                          $(0),AREA2   RES      10
000015                                                      END
            00 000022    000001
            00 000023    000001
            00 000010    000002
            00 000024    201400
            00 000025    000000
            01 000012    000007
            01 000000    000012
```

Explanation:

- Line 2 generates a literal constant 1 under location counter 0.
- Line 3 defines a literal table TAB1 under location counter 0.
- Line 4 generates another literal constant 1, but different from that generated by line 2 because different literal tables are used.
- Line 5 defines a literal table TAB2 starting at address AREA2 under location counter 0 (see line 14). The location counter specification is not used and is superfluous.
- Line 6 generates a literal constant 2 at AREA2.
- Line 7 generates a literal constant 1.0 (2 words) under location counter 0.
- Line 8 uses a lit directive to generate further literals of the type (LITERAL) under location counter 1.

- Line 9 generates a literal constant 7 under location counter 1.
- Line 10 defines literal table AREA1 under location counter 1 (see line 13).
- Line 11 refers to the same literal as line 7. Because the literal was defined previous to the LIT in line 10, it is generated at the end of location counter 1.
- Line 12 generates a literal constant 10 at AREA1.
- Line 13 and 14 reserve 10 words each for the literal tables AREA1 and AREA2.

2.17. INFO DIRECTIVE

The INFO directive provides a means of organizing coding assembled under various location counters into certain system-defined groups. There are six possible groups into which part or all of a program may be divided:

0 – bay-dependent
1 – bay-independent
2 – drum
3 – FASTRAND mass storage
4 – common, bay-independent
5 – common, bay-dependent

- Group 0 – bay-dependent

  Group 0 consists of relocatable object code (instructions and/or constants) written in such a way that it can be relocated anywhere within a bay starting at an even address. If a program of this category exceeds 4096 words (one bay), loading and/or relocation of that program starts automatically at the beginning of a bay. If the size of the location counter is less than 4096 words, all words are allocated within one bay. Since group 0 is the most commonly used relocation mode, it is the assumed group in the absence of an INFO directive for any location counter.

- Group 1 – bay-independent

  Group 1 consists of relocatable object code (mostly constants and some instructions) written in such a way that it can be allocated any available storage location regardless of bay boundaries.

- Group 2 – drum

  This group is used to reserve drum area in 512 18-bit word increments at assembly time and to convey this information to the job loader. It eliminates the need for writing supervisor calls for drum buffer requests and has the added advantage of being processed by the job loader prior to loading the program. If for any reason sufficient drum area is not available, the program is not loaded until sufficient drum space becomes available. Drum space is allocated in such a way that the requested area under each location counter is contiguous unless part of the space is already allocated through an @ASG control card. Location counters of this type may not be used to generate relocatable object code.

  In referencing the drum space allocated through the INFO directive, the location counter is used by the loader as the logical file number. If multiple elements within a single program reserve drum space in this manner, the space is allocated only once for the largest requested area for each location counter (file number).

- Group 3 – FASTRAND mass storage

  Logically, the purpose of this group is the same as that of group 2. Hardware characteristics, however, dictate that FASTRAND allocation is kept separate from drum allocation. A FASTRAND increment is 3584 18-bit words or 1 track.

- Group 4 – common, bay-independent

  This group is simply an extension of group 1, the bay-independent group. It allows separately assembled routines to share storage areas by using the same label in the INFO directives for this group. This capability is provided by the job loader which allocates storage only once for all the references of this label in the routines to be loaded for a program. The length of the storage area is chosen by the job loader to be equal to the longest of the location counter lengths.

- Group 5 – common, bay-dependent

  This is an extension of group 0, the bay-dependent group. It allows separately assembled routines to share storage area by using the same label in the INFO directive for this group. This capability is provided by the job loader which allocates storage only once for all the references of this label in the routines to be loaded for a program. The length of the storage area is chosen by the job loader to be equal to the longest of the location counter lengths.

The symbolic format of the INFO directive is:

label   INFO   g I_1,I_2,I_n

where label is an optional symbolic label which is only meaningful to the job loader. In the case of groups 4 and 5, the label is the common block name. Labels are allowed for groups 0, 1, 2, and 3. The operand field consists of two lists of expressions. The first list represents one of the six group numbers and may consist of one expression only. The second list may consist of one or more expressions, each defining one of the specific location counters assigned to that group.

The assembler allows a maximum of 16 INFO statements which are collected and passed to the job loader.

Example:

```
000001                                 /*
000002                                 COMMON     INFO      4  1,8
000003                                 $(1)
000004          01 000000              INTABL     RES       500
000005          08 000000              $(8),TAPE1   RES     512
000006          08 001000              TAPE2        RES     512
000007          08 002000              TAPE3        RES     1024
000008          08 004000              DRBUF*       RES     256
000009          08 004400              ARRAY*       RES     1000
000010                                            INFO      2  6
000011          06 000000              $(6),DBUF  RES       1000//512
000012          06 000002              DBUF2      RES       2000//512
000013                                 $(0)
000014                                            END

                                          *** SUMMARY ***

   PROGRAM SIZE:   01 00764    06 00006    08 06350

   EXTERNAL DEFINITIONS:               DRBUF    ARRAY
```

Explanation:

- Line 2 specifies that the code generated under location counters 1 and 8 is to be considered as bay-independent common storage (group 4). The common area is identified by the name COMMON.
- Lines 3 through 9 define various buffers in the common area.
- Lines 10, 11, and 12 specify that six blocks of drum space are to be allocated. The label DBUF refers to the drum address of the first block of this drum area. The label DBUF2 refers to the drum address of the third block of this drum area.

2.18. ASM DIRECTIVE

The ASM directive is not an assembler directive; it is a library procedure. The procedure may be used to generate a series of data words (or instructions) in one statement. The format is:

label ASM $e_1, e_2, e_3, \ldots, e_n$

The label, if present, refers to the first data word generated, e_1. The operand consists of a series of expressions e_i each of which is generated as one or more data words.

Example:

```
000001                                             /*
000002             00 000000   000001                          ASM   1,2,+(LLK 1),+LABEL,'ABCD'
                   00 000001   000002
                   00 000002   70 0001
            U      00 000003   000000
                   00 000004   242526
                   00 000005   270000
000003                                                         END

                                                  *** SUMMARY ***

PROGRAM SIZE:   00 00006

EXTERNAL OR UNDEFINED REFERENCES:   LABEL
```

Explanation:

The code generated by the ASM procedure call is equivalent to the series of statements:

```
+1
+2
LLK 1
+LABEL
'ABCD'
```

The ASM procedure is illustrated in 3.7.2.

3. PROCEDURES

3.1. GENERAL

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. A procedure may be generated any number of times with different parameters supplied each time it is referenced. Procedures are implemented by the PROC directive. The source code between the PROC and END directives is commonly referred to as the procedure sample. 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 unless it is defined in the library as a PROC element.

3.2. PROCEDURE MODES

Procedures can be developed in any of three modes: simple, generative, or interpretive. The differences between simple, generative, and interpretive procedures are functional differences only, not intrinsic in the manner in which the assembler analyzes them. Many procedures are actually combinations of all of them.

3.2.1. Simple Mode

The simple mode occurs when the object procedure developed is equivalent to the object procedure declared. In this mode, the procedure is used essentially to provide program legibility and avoid repetition of code. An example of a simple mode procedure is given in 3.7.3.

3.2.2. Generative Mode

The generative mode occurs if the object procedure developed is a multiple of the object procedure defined. By combining the DO directive and a simple mode procedure, the same code may be generated a number of times. An example of a generative procedure is given in 3.7.2.

3.2.3. Interpretive Mode

The interpretive mode occurs when the object procedure determines which code is to be generated, based on the parameters supplied when the procedure is called. In this mode, the PROC body provides the algorithms to be used for the generation of code. Examples of interpretive procedures are given in 3.7.4, 3.7.5, and 3.7.6.

3.3. PROCEDURE SAMPLE

A procedure sample consists of a group of statements having a PROC and an END directive as delimiters. The procedure sample is stored by the assembler so that it may be scanned when the procedure is called upon as a result of the occurrence of one of its entry points in the function field. The procedure sample is scanned at least once for each time it is called upon.

3.4. PROC DIRECTIVE

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 one of the means by which the procedure may be referenced.

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

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

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

- if the procedure can generate a variable number of words;
- if forward references are made in the procedure;
- if external definitions are made in the procedure;
- when a label on a procedure reference line is to be assigned to a line other than the first line of the procedure;
- when a procedure call is present in the procedure which causes the assembler to bring the second procedure from the library into the procedure storage area.

Except for the foregoing conditions, subfield 2 should be used because it eliminates two subassembly passes of the procedure sample, thereby shortening assembly time.

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

Example:

LABEL	OPERATION	OPERAND
CØMPAR	PRØC	1,10
MØVE	PRØC	.

Explanation:

- Line 1 contains the label COMPAR. Subfield 1 of the operand specifies that one field may appear on the reference line. Subfield 2 indicates that ten words are generated by the procedure.
- Line 2 has no operand field.

3.5. 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:

LABEL	OPERATION	OPERAND
LØAD*	PRØC	0,1
	LL	TAG
	END	

Explanation:

- Lines 1, 2, and 3 define the procedure sample.
- Line 1 specifies that no parameters are supplied on the call line, that one word is to be generated whenever the procedure is called, and that the entry point to the procedure is LOAD.
- Line 2 contains the instruction LL TAG which is to be generated each time the PROC is called.
- Line 3 specifies the end of the procedure sample.

3.6. PROCEDURE REFERENCE

When a procedure reference is encountered at assembly time, the specified procedure sample is analyzed. If the procedure sample is contained within the assembled program, it must be defined prior to the first reference. If the procedure sample is defined in a procedure element in the user or system library, the entire PROC element will be included in the assembler PROC storage area when a call on any one of its procedures is made. In searching the libraries for a procedure entry point, the user library is searched first. Since the entire procedure element is inserted by a reference on one of its PROCs, care must be taken that no duplication of procedure entry points occurs when multiple PROC elements are inserted. To reference a procedure, a call line is used.

3.6.1. 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 procedure label operand

The label field of a call line is optional.

The operation field contains the entry point 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.

Example:

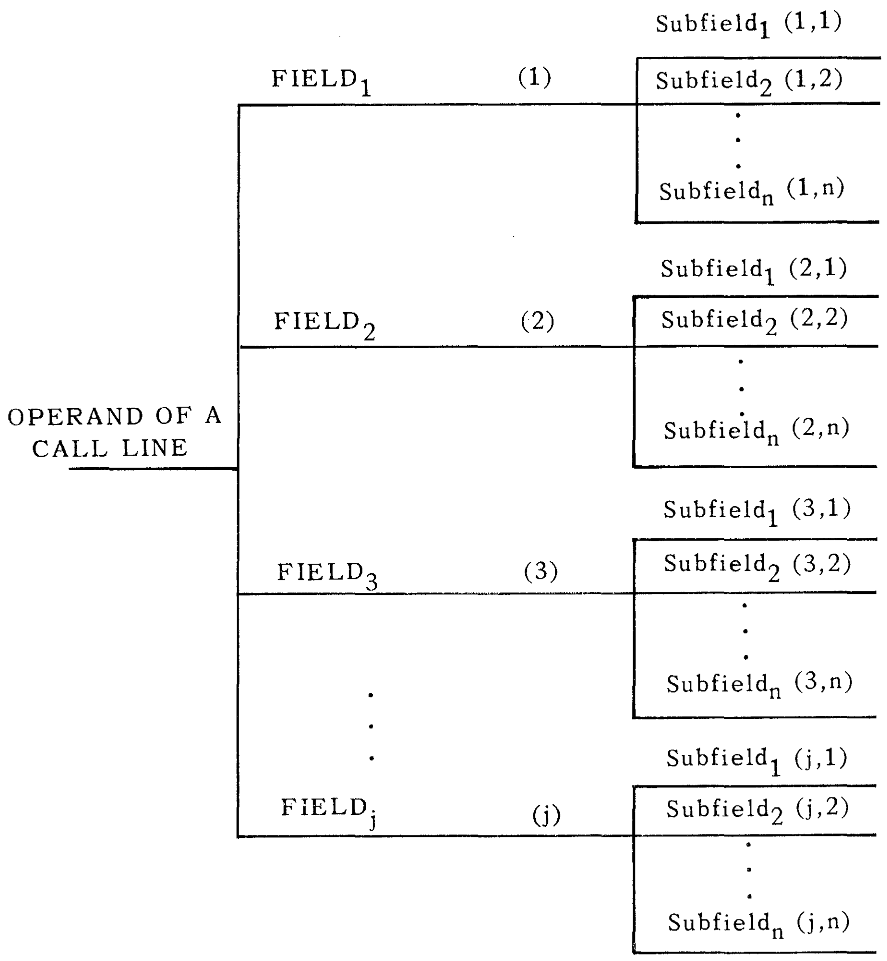
```
   LABEL          OPERATION                 OPERAND
1          10               20               30         40
          LØAD
CALL1     SPEC .
ADDP      ADD22     4,TAG   99,PUR .
```

Explanation:

- Line 1 has no label and the procedure LOAD will be generated.
- Line 2 contains the label, CALL1. The procedure referenced is SPEC.
- Line 3 contains the label ADDP. The procedure ADD22 is referenced. The operand field contains four parameters. The parameters supplied are grouped into two fields with two subfields each.

3.6.2. 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. The 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:

```
LABEL          OPERATION            OPERAND
1         10             20         30         40
          LA             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.

3.7. PARAFORMS

The parameter reference form, commonly called the paraform, provides a means for selectively referring to the operand parameters of a procedure call line. Paraforms are implicitly defined by the operand field parameters of the procedure call line. They are used in the operand field of a line of symbolic coding within the procedure sample. Paraforms are only defined during the processing of the procedure call line and the referenced procedure sample.

A paraform is identified by the name of the procedure reference. There are six syntactical paraform structures which denote different values associated with the operand field parameters of the procedure call line.

3.7.1. Referencing the Number of Fields

When the procedure name is used in the operand field of a symbolic line within the procedure sample, it is equated to a constant equal to the number of fields in the call line.

Example:

```
000001                                      /.
000002                                      CALL1*   PROC      .
000003                                      I        DO        CALL1 , RES  1
000004                                               END
000005                                      .
000006                                      . ***    CALL NAME
000007                                      .
000008          00 000000                            CALL1     FIRST SECND THRL
                00 000001
                00 000002
000009          00 000003    000001         FIRST    +1
000010          00 000004    000002         SECND    +2
000011          00 000005    000003         THRD     +3
000012                                               END
```

Explanation:

- Lines 2 and 4 define the procedure CALL1.
- Line 3 reserves one word CALL1 times.
- Line 8 calls procedure CALL1 with three fields, FIRST, SECND, and THRD. As a result the paraform reference CALL1 in line 2 is assigned the value 3 and three words are reserved.

Example:

```
000001                                        /.
000002                                        ADD*     PROC     .
000003                                                 LL       ADD(1,1)
000004                                                 AL       ADD(1,2)
000005                                                 DO       ADD=2 , SL  ADD(2,1)
000006                                                 END
000007                                        .
000008                                        . ***    CALL NAME
000009                                        .
000010         00 000000    12 0003                    ADD      A,B     C
               00 000001    14 0004
               00 000002    44 0005
000011         00 000003    000001            A        +1
000012         00 000004    000002            B        +2
000013         00 000005    000003            C        +3
000014                                                 END
```

Explanation:

- Lines 2 through 6 define the procedure ADD.
- Lines 3 and 4 define a simple addition.
- Line 5 contains a conditional DO statement. The condition is dependent on the number of fields in the call line, in this case two.
- Line 10 is the call line consisting of two fields.

3.7.2. Referencing the Number of Subfields

The paraform pn (a), where pn denotes the procedure name and (a) is an expression which represents the a^th field on the procedure call line, refers to the number of subfields present in the a^th field.

Example:

```
000001                                        /.
000002                                        ASM*     PROC .
000003                                        I        DO    ASM(1) , +ASM(1,I)
000004                                                 END
000005                                        .
000006                                        . ***    CALL NAME
000007                                        .
000008         00 000000    000001                     ASM        1,2,4
               00 000001    000002
               00 000002    000004
000009                                                 END
```

Explanation:

- Lines 2 and 4 define the procedure ASM.
- Line 3 performs the operation +ASM(1,I), ASM(1) times.
- Line 8 calls the procedure ASM and specifies one field with three subfields 1, 2, and 4. As a result the paraform ASM(1) is assigned the value 3, and the operation +ASM(1,I) is performed three times. The code generated as a result of the ASM call will therefore be three data words:

  +1

  +2

  +4

3.7.3. Referencing the Procedure Call Parameters

In order to reference any of the supplied procedure parameters, the specific parameter is identified by specifying the procedure name immediately followed by a pair of parentheses. Enclosed within the parentheses are two values separated by a comma. The first value denotes the specific field in the call line; the second value denotes the specific subfield within the specified field in the call line.

Example:

```
000001                                     /.
000002                                     ADD*      PROC      1,3
000003                                               LL        ADD(1,1)
000004                                               AL        ADD(1,2)
000005                                               SL        ADD(1,3)
000006                                               END
000007                                     .
000008                                     . ***     CALL NAME
000009                                     .
000010      00 000000    12 0003                     ADD       ONE,TWO,THREE
            00 000001    14 0004
            00 000002    44 0005
000011      00 000003    000001            ONE       +1
000012      00 000004    000002            TWO       +2
000013      00 000005    000003            THREE     +3
000014                                               END
```

Explanation:

- Lines 2 and 6 define the procedure ADD.
- Lines 3, 4, and 5 generate a load, add, store set of instructions to perform the operation C = A+B. The addresses of A, B, and C are specified as subfields 1, 2, and 3 of field 1.
- Line 10 calls upon the procedure ADD to generate the code which performs the operation (ONE)+(TWO)→THREE.

3.7.4. Referencing the Asterisk in a Procedure Parameter

Because of the use of the asterisk to indicate the index mode in normal instructions, the presence or absence of an asterisk in the procedure call parameter may be checked by using the paraform structure pn(a,*b), where pn denotes the procedure name, and a and b are expressions representing the field and subfield numbers respectively. If the specified parameter, pn(a,b), is preceded by an asterisk, the paraform pn(a,*b) is assigned a value 1;otherwise, 0.

Example:

```
000001                                            /.
000002                                            L          PROC       1
000003                                            MOVE*      NAME
000004                                                       DO         L(1,*1) , LU L(1,1)
000005                                                       DO         L(1,*2) , LL L(1,2)
000006                                                       BT         L(1,3)
000007                                                       END
000008                                            .
000009                                            .  ***     CALL EXAMPLE
000010                                            .
000011       00 000000     10 0036                           MOVE       *(FROM),*(TO),12
             00 000001     12 0037
             00 000002     5070 14
000012       00 000003     10 0036                           LU         (FROM)
000013       00 000004     12 0037                           LL         (TO)
000014       00 000005     5070 14                           MOVE       0,0,12
000015       00 000006                            FROM       RES        12
000016       00 000022                            TO         RES        12
000017                                                       END
             00 000036     000006
             00 000037     000022
```

Explanation:

- Lines 2 and 7 define the procedure L.
- Line 3 defines MOVE as an entry point to the procedure.
- Lines 4 and 5 generate the instructions LU L(1,1) and LL L(1,2) if the first and second subfields of the first field of the parameters on the call line are preceded by an asterisk.
- Line 11 calls on procedure L by way of the entry point MOVE. Since asterisks occur in the first two subfields, an LU (FROM) and LL (TO) are generated.
- Line 14 calls on the MOVE procedure. Since no asterisk occurs on the first two parameters, the LU and LL are not generated.

Example:

```
000001                                     L       PROC    1,2 *
000002                                     LA*     NAME
000003                                     I       FORM    6,12
000004                                             I       010*L(1,*1),L(1,1)
000005                                             I       012*L(1,*1),L(1,1)+1
000006                                             END
000007   U     00 000000   10 0000                 LA      TAG
         U     00 000001   12 0001
000008   U     00 000002   11 0000                 LA      *TAG
         U     00 000003   13 0001
000009                                             END
```

Explanation:

- Lines 1 and 6 define procedure L.
- Line 2 defines an entry point LA. (See 3.6.2).
- Lines 4 and 5 generate:

  (1) lower 12 bits equal to the supplied first parameter

  (2) upper 6 bits 010 (LU) and 012 (LL) if no asterisk precedes the first parameter

  (3) upper 6 bits 011 (LU*) and 013 (LL*) if an asterisk precedes the parameter

- Line 7 causes the following instructions to be generated:

  LU TAG

  LL TAG+1

- Line 8 causes the following instructions to be generated:

  LU *TAG

  LL *TAG+1

3.7.5. Referencing the NAME Directive Operand Value

The NAME directive may define a procedure entry point (see 2.14). The paraform pn(0,0), where pn denotes the procedure name, refers to the value in the operand field of the NAME directive by which the procedure was called upon. If the procedure call is to the procedure name itself, pn(0,0) has a value of 0.

Example:

```
000001                                       /.
000002                                       L         PROC     1,3
000003                                       ADD*      NAME     014                       . 'AL' FCN. CDE
000004                                       SUB*      NAME     016                       . 'ANL' FCN. CDE
000005                                       I         FORM     6,12
000006                                                 I        012+L(1,*1),L(1,1)
000007                                                 I        L(0,0)+L(1,*2),L(1,2)
000008                                                 I        044+L(1,*3),L(1,3)
000009                                                 END
000010                                       .
000011                                       . ***     CALL NAME
000012                                       .
000013        00 000000     12 0006                    ADD      ONE,*TWO,THREE
              00 000001     15 0007
              00 000002     44 0010
000014        00 000003     12 0011                    SUB      A,B,C
              00 000004     16 0012
              00 000005     44 0013
000015        00 000006     000001           ONE       +1
000016        00 000007     000002           TWO       +2
000017        00 000010     000003           THREE     +3
000018        00 000011     000004           A         +4
000019        00 000012     000005           B         +5
000020        00 000013     000006           C         +6
000021                                                 END
```

Explanation:

- Lines 2 and 9 define the procedure L.
- Lines 3 and 4 provide the entry points ADD and SUB. (Note that L is not an entry point to the procedure since no asterisk is appended to the label.) If the procedure is called upon through the entry point ADD, the value of L(0,0) is 014; if called upon by way of the entry point SUB, L(0,0) is 016.
- Lines 6, 7, and 8 generate the instructions:

  LL or LL*

  AL,ANL or AL*,ANL*, and

  SL or SL*, respectively.

  The indexed function codes a:e used if an asterisk precedes the appropriate paraform expression. Depending on whether the ADD or SUB entry point is used, the AL or ANL function code is used.

- Line 13 calls on procedure L through the entry point ADD. As a result, the code generated is:

  LL ONE

  AL *TWO

  SL THREE

- Line 14 calls on procedure L through the entry point SUB. As a result, the generated code is:

  LL A

  ANL B

  SL C

3.7.6. Referencing Subfields of the 0th Field

The paraform pn(0,b), where pn represents the procedure name, may be used to denote the bth subfield of the 0th field. The 0th field is defined on the procedure call immediately following the procedure call name and separated by a comma.

Example:

```
000001                                              /*
000002                                              L         PROC      1
000003                                              ADD*      NAME      014
000004                                              SUB*      NAME      016
000005                                              I         FORM      6,12
000006                                                        DO  L(0,1)>0 , I 012+L(1,*1),L(1,1)
000007                                                        I         L(0,0)+L(1,*2),L(1,2)
000008                                                        DO  L(0)=2 , I 044+L(1,*3),L(1,3)
000009                                                        END
000010                                              .
000011                                              . ***     CALL NAME
000012                                              .
000013           00 000000    14 0006                         ADD       A,B,*C
000014           00 000001    12 0005                         SUB,1     A,B,*C
                 00 000002    16 0006
000015           00 000003    14 0006                         ADD,0,1   A,B,*C
                 00 000004    45 0007
000016           00 000005    000001                A         +1
000017           00 000006    000002                B         +2
000018           00 000007    000003                C         +3
000019                                                        END
```

Explanation:

- Lines 2 and 9 define the procedure L.
- Lines 3 and 4 provide the entry points ADD and SUB.
- Line 6 generates an LL or LL* instruction if the first subfield of the 0th field is present and greater than zero.
- Line 7 generates the instruction AL or ANL, depending on the entry point used.
- Line 8 generates an SL or SL* instruction if the second subfield of the 0th field is present.
- Line 13 generates the code:

  AL B

  (Note that subfields L(1,1) and L(1,3) are present but superfluous).

- Line 14 generates the code:

```
LL   A
ANL  B
```

(Note that the subfield L(1,3) is superfluous.)

- Line 15 generates the code:

```
AL  B
SL  *C
```

(Note that even though subfield L(1,1) must be present, no actual expression is needed. A zero would suffice to define the subfield).

3.7.7. Summary of Paraforms

Paraform constructions are summarized as follows (pn denotes procedure name):

pn	When the procedure name is written with no specified field or subfield, the value of the paraform is a constant equal to the number of fields in the call line. (The operation field is not included as part of the count.)
pn(a)	The value of pn(a) is a constant equal to the number of subfields in the specified (a) field.
pn(a,b)	The value of pn(a,b) is the parameter appearing in the subfield of field a.
pn(a,*b)	The value of pn(a,*b) is a constant equal to 1 or 0, depending on whether the parameter in the bth subfield of the ath field is preceded by an asterisk.
pn(0,0)	The paraform pn(0,0) has a value equal to that specified in the operand field of the NAME directive used for the procedure call entry point. If the entry point is the procedure name itself, pn(0,0) has a constant value equal to 0.
pn(0,b)	The paraform pn(0,b) has a value equal to the parameter in the bth subfield of the 0th field. The 0th field is considered to be the operation field.

3.8. NESTING OF PROCEDURES

When encountering a procedure call, the assembler temporarily discontinues the current assembly and begins a subassembly of the procedure sample. Upon encountering the END directive, the original assembly is resumed. While processing the procedure sample, another procedure call may be encountered, resulting in the temporary suspension of the first procedure and the processing of the second. This process may continue up to 15 levels of procedures and is referred to as the nesting of procedures. Each time a subassembly of a procedure is entered, all labels within the procedure are defined for that procedure only. All labels and paraforms defined in all preceding assemblies are also available to the subassembly. When the main assembly is resumed, all labels defined within the subassembly are erased.

The nesting of procedures, therefore, enables the programmer to use the same label in different procedures. Nesting allows simpler block-building techniques but requires longer assembly time.

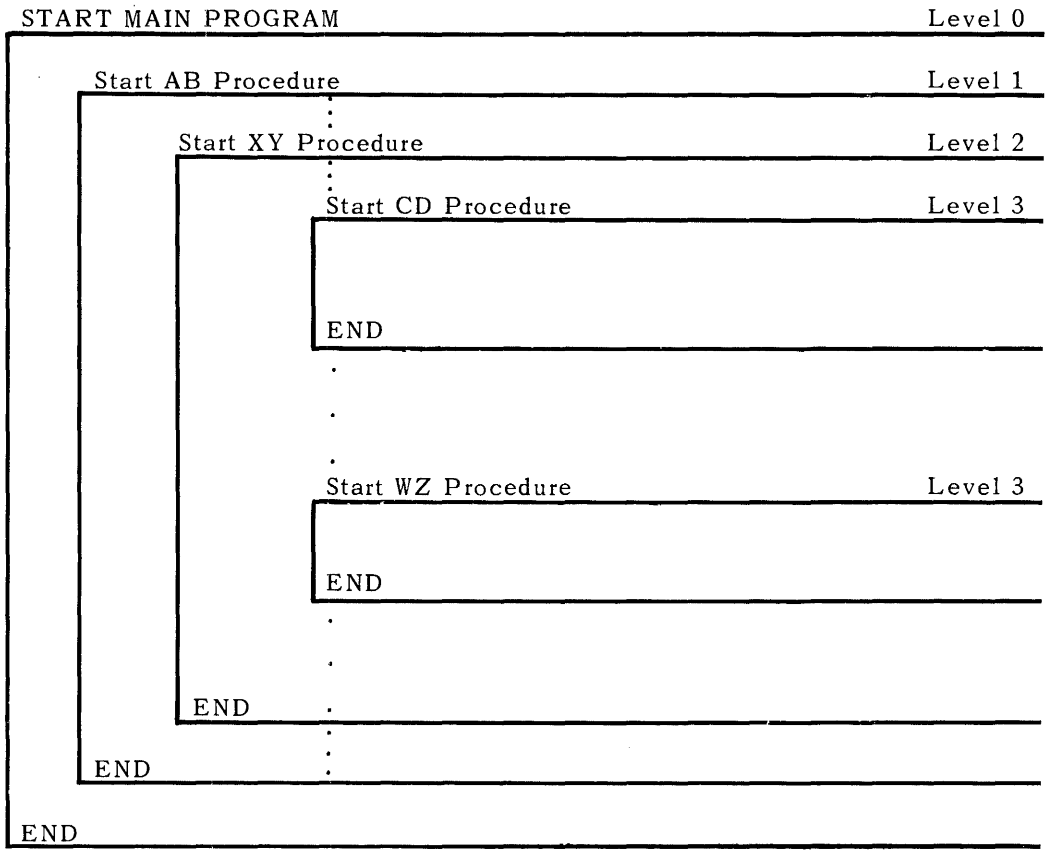
When practical, the depth of nesting should be limited. Use of the distributed NAME and GO directives may be helpful in restricting levels of nesting (see 2.13 and 2.14).

3.8.1. 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 defined at one level higher than the external procedure. Procedures may be nested to 15 levels. Therefore, the physical location of the procedure sample determines at which level the procedure can be accessed.

Physical nesting of procedures may be used to prevent certain procedures from being referenced unconditionally.

Example:



Explanation:

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

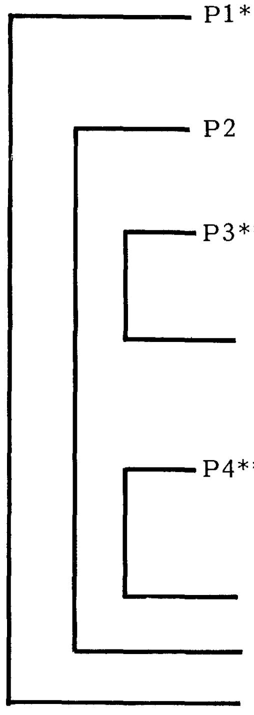
3.8.2. Levels of Procedures

When procedures are nested, they are considered to have various levels of hierarchy. The main program is considered level 0. A procedure called upon at level 0 is assembled at level 1. Its entry point must therefore be defined to be accessible to level 0. A procedure called upon within a level 1 procedure is assembled at level 2. In other words, each time a new subassembly is started the level is increased by 1, and decreased as the procedure subassembly is completed.

The level of a procedure entry point determines where the procedure may be referenced. If the level of the procedure entry point is equal to or less than the level of the subassembly, it is accessible to that subassembly, and the procedure may be referenced. If the level of the procedure entry point is greater than the level of the subassembly, the procedure may not be referenced from within the subassembly.

The level of a procedure entry point is determined by combining the level at which the procedure sample is defined and the number of asterisks appended to the label of the entry point. Each asterisk appended to the label of the procedure entry point makes the label accessible for reference at a level one lower than the level at which the procedure sample is defined.

Example:



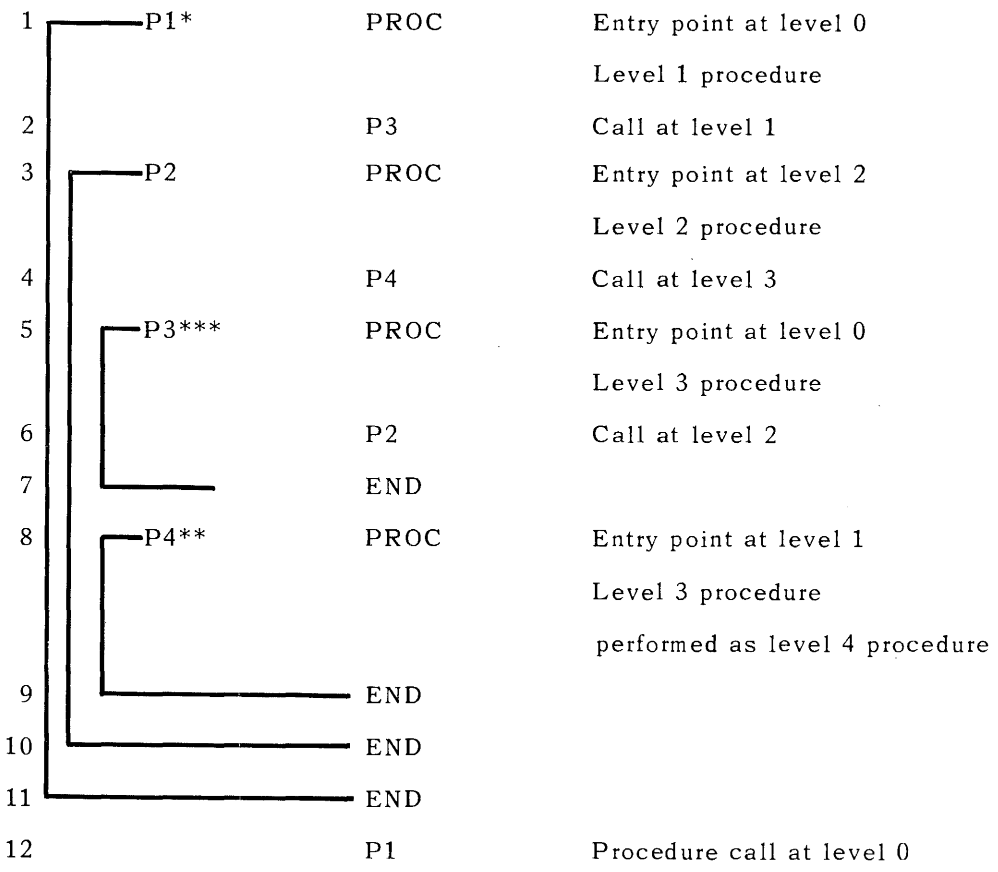
```
P1*        PROC     Entry point at level 0
                    Level 1 procedure
P2         PROC     Entry point at level 2
                    Level 2 procedure
P3***      PROC     Entry point at level 0
                    Level 3 procedure
           END

P4**       PROC     Entry point at level 1
                    Level 3 procedure
           END
           END
           END
           Level 0 code
```

Explanation:

- Entry point P1 is accessible to level 0. Procedure P1 may be called from anywhere in the program.
- Entry point P2 is accessible to level 2. Procedure P2 may be called only from within a second or higher level procedure.
- Entry point P3 is accessible to level 0. Procedure P3 may be called from anywhere in the program.
- Entry point P4 is accessible to level 1. Procedure P4 may be called from within a first or higher level procedure only.

Example:



```
1    P1*       PROC    Entry point at level 0
                       Level 1 procedure
2              P3      Call at level 1
3    P2        PROC    Entry point at level 2
                       Level 2 procedure
4              P4      Call at level 3
5    P3***     PROC    Entry point at level 0
                       Level 3 procedure
6              P2      Call at level 2
7              END
8    P4**      PROC    Entry point at level 1
                       Level 3 procedure
                       performed as level 4 procedure
9              END
10             END
11             END
12             P1      Procedure call at level 0
```

Explanation:

- Lines 1 and 11 define a level 1 procedure P1.
- Lines 3 and 10 define a level 2 procedure P2.
- Lines 5 and 7 define a level 3 procedure P3.
- Lines 8 and 9 define a level 3 procedure P4.

- Line 1 defines a procedure entry point P1 at level 0.
- Line 3 defines a procedure entry point P2 at level 2.
- Line 5 defines a procedure entry point P3 at level 0.
- Line 8 defines a procedure entry point P4 at level 1.
- Line 12 is a procedure call on procedure P1 which is accessible at all levels. The procedure P1 is processed at level 1.
- Line 2 is a procedure call on procedure P3 which is accessible at all levels. The procedure P3 is processed at level 2.
- Line 6 is a procedure call on procedure P2 which is accessible at level 2 and higher. The procedure P2 is processed at level 3.
- Line 4 is a procedure call on procedure P4 which is accessible at level 1 and higher. The procedure P4 is processed at level 4.

3.9. PROCEDURE LABELS

As stated previously, the labels on the PROC and NAME directives are procedure entry points. They may be referenced as procedure entry points only at those levels or higher levels of subassembly at which the entry point is defined. They are inaccessible below the level at which the entry point is defined. The accessible level of the entry point is determined by the physical nesting depth of the procedure together with the number of asterisks appended to the entry point label.

Other labels may be used within procedures. A label is a symbolic representation of some value. It may be local or global. A local label may be referenced only at the level at which it is defined or at higher levels. A global label is one which is defined to be accessible beyond the range of the assembly in which it is defined. When a label is defined to be accessible beyond the entire assembly, it is said to be externally defined.

Labels defined in the main program may therefore be referenced within any procedure. Labels defined within a particular procedure may normally be only referenced within that procedure or by any procedure called upon by the first procedure.

Example:

```
000001                                           /*
000002                                           A*        PROC
000003                                           ONE       EQU       1
000004                                           B*        PROC
000005                                                     LLK       ONE
000006                                           TWO       EQU       2
000007                                                     LLK       TWO
000008                                                     LLK       THREE
000009                                                     END
000010                                                     LLK       ONE
000011                                                     B
000012                                                     LLK       THREE
000013                                                     END
000014                        000003             THREE     EQU       3
000015                                           *
000016                                           * ***     CALL NAME
000017                                           *
000018             00 000000  70 0001                      A
                   00 000001  70 0001
                   00 000002  70 0002
                   00 000003  70 0003
                   00 000004  70 0003
000019             00 000005  70 0003                      LLK       THREE
000020                                                     END
```

Explanation:

- Lines 2 and 13 define a first level procedure, A.
- Lines 4 and 9 define a second level procedure, B.
- Line 3 defines ONE at level 1.
- Line 6 defines TWO at level 2.
- Line 14 defines THREE at level 0.
- Lines 5, 7, and 8 illustrate that all three labels may be referenced within the second level procedure B.
- Lines 10 and 12 illustrate that only the labels ONE and THREE may be referenced in the first level procedure A.
- Line 19 illustrates that only the label THREE may be referenced in the main program. The labels ONE and TWO are not defined to be accessible to level 0.

Labels defined within a procedure are unique to the level at which they are defined. If the same label is defined at more than one level, any reference to that label will be to the definition in existence at the highest accessible level.

Example:

```
000001                                      /.
000002                                      A*      PROC
000003                                      ONE     EQU     1
000004                                      B*      PROC
000005                                      ONE     EQU     2
000006                                              LLK     ONE             . ONE = 2
000007                                              END
000008                                              B
000009                                              LLK     ONE             . ONE = 1
000010                                              END
000011                   000003             ONE     EQU     3
000012                                      .
000013                                      . ***   CALL NAME
000014                                      .
000015        00 000000  70 0002                    A
              00 000001  70 0001
000016        00 000002  70 0003                    LLK     ONE             . ONE = 3
000017                                              END
```

Explanation:

- Lines 2, 4, 7, and 10 define the first and second level procedures A and B.
- Lines 3, 5, and 11 define ONE as 1, 2, and 3, respectively, at levels 1, 2, and 0.
- Lines 6, 9, and 16 illustrate that even though the same label ONE is used, the values associated in each case are different.

NOTE: If line 5 were omitted, the reference to ONE in line 6 would result in a reference to the value of ONE defined at the next lower level, namely 1.

3.9.1. Global Labels

In order to define labels to be accessible at levels lower than the one at which they are defined, asterisks are appended to the label definition. For each asterisk appended to the label, the level of the label is decremented by 1. If the number of asterisks appended to the label definition exceeds the subassembly level at which it is defined, the label becomes an external definition and may be referenced by other programs.

Global labels are defined only after the procedure in which they are defined has been called.

Care must be taken that global labels are not multiply defined as a result of repeated calls on the procedure in which they are defined.

Example:

```
000001                                               /*
000002                                               BREGS*     PROC
000003                                               B1**       EQU       1
000004                                               B2**       EQU       2
000005                                               B3*        EQU       3
000006                                                          END
000007                                               *
000008                                               * ***      CALL NAME
000009                                               *
000010                                                          BREGS
000011           00 000000   32 0001                            LB        B1
000012                                                          END

                                                         *** SUMMARY ***

     PROGRAM SIZE:   00 00001

     EXTERNAL DEFINITIONS:                  B2        B1
```

Explanation:

- Lines 2 and 6 define the procedure BREGS.
- Line 10 calls on the procedure BREGS and causes the labels B1, B2, and B3 to be defined.
- Lines 3 and 4 define the external labels B1 and B2 as external definitions.
- Line 5 defines B3 = 3 at level 0.
- Line 11 illustrates that after the procedure BREGS is called, the label B1 may be referenced at level 0.

3.10. FORWARD REFERENCES

Forward references occur when a label is referenced prior to its definition. Forward references also occur if a label whose value is dependent upon values not yet defined has been referenced. Forward references are prohibited if the fact that different values associated with the label in pass 1 and pass 2 of the assembly causes different amounts of code to be generated in pass 1 and pass 2 of the assembly.

Example:

LABEL	OPERATION	OPERAND
	RES	A
A	EQU	5
	DO B>0 ,	LLK 1
B	LL	A

Explanation:

- A is not defined in pass 1.
- B is not defined in pass 1.

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 chooses an error exit on pass 1 (that is, no generation of code) and produces code on pass 2, the labels following the call on the sample are assigned a location counter value on pass 1 that is different in pass 2. The result is a multiple definition of those labels.

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

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

3.11. LOCATION COUNTER DEFINITION

A procedure may be made to generate code under one or more location counters by defining the location counter in the label field of a line item within the procedure sample. When the procedure is completed, the location counter active at the time that the procedure was called is reactivated.

Example:

```
000001                                              /*
000002                                              A*        PROC
000003                                                        SLJI      $(1)
000004                                              $(1)      +A(1,1)
000005                                                        END
000006                                              *
000007                                              * ***     CALL NAME
000008                                              *
000009            00 000000   30 0000                         A         SUB1
                                + 6,12 01
          U       01 000000   000000
                                +    18 00      SUB1
000010            00 000001   30 0001                         A         SUB2
                                + 6,12 01
          U       01 000001   000000
                                +    18 00      SUB2
000011                                                        END

                                                         *** SUMMARY ***

   PROGRAM SIZE:   00 00002     01 00002

   EXTERNAL OR UNDEFINED REFERENCES:   SUB2      SUB1
```

Explanation:

- Lines 2 and 5 define the procedure A.
- Line 3 generates an SLJI instruction to the next word under location counter 1.
- Line 4 defines the subroutine entry address under location counter 1.
- Lines 9 and 10 generate two calls on subroutines SUB1 and SUB2 respectively. The SLJI instructions are generated under location counter 0; the entry point addresses, SUB1 and SUB2, are generated under location counter 1.

  NOTE: Unless a map is submitted to force location counters 0 and 1 to be in the same bay, the foregoing example is not executable.

3.11.1. Writing 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 has appeared in place of the asterisk except that it is defined at the level of the reference line on which it appeared.

Example:

```
000001                                    /.
000002                                    A*      PROC      1,2
000003                                            TZ        A(1,1)
000004                                    *       J         A(1,2)
000005                                            END
000006                                    *
000007                                    * ***   CALL NAME
000008                                    *
000009       00 000000    57 0003         JP      A         ONE,TWO
             00 000001    34 0004
000010       00 000002    55 0001                 JI        JP
000011       00 000003    000001          ONE     +1
000012       00 000004    000002          TWO     +2
000013                                            END
```

Explanation:

- Lines 2 and 5 define the procedure A, which generates the instructions TZ and J.
- Line 4 generates the instruction J and has a single asterisk in the label field.
- Line 9 calls on procedure A. The label JP is defined as equal to the location counter value of the J instruction instead of the usual TZ instruction which is the first line generated.

3.12. Complex Procedures

The following paragraphs of this section contain a discussion of those assembler features which enable the construction and use of complex procedures. When the DO, NAME, and GO directives are used in conjunction with procedures, a powerful tool exists for the generation of code which is conditioned by the supplied parameters. Procedures may be used to conditionally generate code. The PROC structure enables coding of generation alogrithms in the procedure sample, such that the code generated applies the alogrithms and may generate entirely different tables or instructions, depending on the supplied parameters.

3.12.1. NAME Directive

The NAME directive has three functions:

- It provides a local reference point within a given procedure sample.
- It provides alternate entry points to the procedure.
- It may supply a value to the procedure which is unique for the associated entry point.

The NAME directive has the structure:

label   NAME   operand

The label field contains a symbolic label no longer than six alphanumeric characters, which is used to identify the NAME directive. The operand field may contain a value which can be referenced in the procedure sample by the paraform pn(0,0), where pn denotes the procedure name.

The label of the NAME directive is defined in the same way as the label of a PROC directive; that is, it is defined at the same level as the procedure, and asterisks are used to make the label accessible at lower levels.

3.12.1.1. Local Reference Point

The NAME directive provides a local reference point within the procedure sample in which it is defined. Associated with the label of the NAME directive is the start of the symbolic code within the procedure immediately following the NAME directive. By using the GO directive (see 3.12.2) or by using the NAME directive as a procedure entry point (see 3.12.1.2), different paths through the procedure sample may be chosen.

3.12.1.2. Alternate Entry Point

The NAME directive may be used as an alternate entry point to the procedure. In this form the same rules applying to the PROC directive entry point apply to the NAME labels. Regardless of the procedure entry point used for any particular procedure call, the paraform name is the procedure name.

Example:

```
000001                                  /.
000002                                  LADD*     PROC
000003                                            LL        LADD(1,1)
000004                                  ADD*      NAME
000005                                            AL        LADD(1,2)
000006                                            END
000007                                  .
000008                                  . ***     CALL NAME
000009                                  .
000010        00 000000    12 0003                LADD      A,B
              00 000001    14 0004
000011        00 000002    14 0004                ADD       0,B
000012                                  .
000013        00 000003    000001       A         +1
000014        00 000004    000002       B         +2
000015                                            END
```

Explanation:

- Lines 2 and 6 define the procedure LADD.
- Line 4 defines the alternate entry point ADD. The entry point ADD, because of its position, does not point to the same procedure sample. If the procedure is called upon through the entry point ADD, the subassembly of the procedure starts with line 5.
- Line 10 calls on the procedure LADD and would generate the instructions:

  LL A

  AL B

- Line 11 calls on the procedure LADD but through entry point ADD. As a result, the generated code would be:

  AL B

3.12.1.3. Parameter Value

The paraform pn(0,0) has a value depending on the procedure entry point used in the the function field of the procedure call line. If the procedure name is used as the entry point, the paraform pn(0,0) has a value of 0. If an entry point defined on a NAME directive is used, the paraform pn(0,0) has a value equal to the operand value of the NAME directive.

Example:

```
000001                                     /.
000002                                     L       PROC    1,2
000003                                     LA*     NAME    010                      . LU FCN. CDE.
000004                                     SA*     NAME    046                      . SU FCN. CDE.
000005                                     I       FORM    6,12
000006                                             I       L(0,0),L(1,1)
000007                                             I       012+032*(L(0,0)=046),L(1,1)+1
000008                                             END
000009                                     .
000010                                     . ***   CALL NAME
000011                                     .
000012    00 000000   10 0004                      LA      A
          00 000001   12 0005
000013    00 000002   46 0006                      SA      B
          00 000003   44 0007
000014                                     .
000015    00 000004   201400               A       +1,0
          00 000005   000000
000016    00 000006                        B       RES     2
000017                                             END
```

Explanation:

- Lines 2 and 8 define the procedure L.
- Lines 3 and 4 provide the entry points LA and SA. If the procedure is called through the entry point LA, the paraform L(0,0) has a value 010; if called through the entry point SA, the paraform L(0,0) has a value 046.
- Line 6 generates an instruction with function codes of either 010 or 046, that is, an LU or an SU.
- Line 7 generates an instruction with function codes of either 012 or 044, that is, an LL or an SL.
- Line 12 calls on procedure L through the entry point LA. The code generated is:

  LU A

  LL A+1

- Line 13 calls on procedure L through the entry point SA. The code generated is:

  SU B

  SL B+1

3.12.2. GO Directive

The GO directive provides a means of transferring control to the line whose label is in the operand field. The format of the GO directive is as follows:

GO   label

The label specified in the operand field must refer to the label of a NAME or PROC directive and must be accessible at the level at which the GO is performed.

When the GO directive is encountered within a procedure, the next symbolic line scanned in the procedure sample is the one to which the NAME directive referenced points. The NAME directive referenced need not be defined in the procedure. As a result, lateral transfer between procedures is possible through the use of the GO directive.

In determining the label of the NAME directive referred to, the assembler uses the following alogrithms:

- If the first character of the operand field of the GO directive is alphabetic, the label is directly specified.

- If the first character of the operand field of the GO directive is not alphabetic, the field is assumed to contain an expression. The resultant 36-bit value of the expression is then used as representing the left-justified label.

Example 1:

```
000001                                        /*
000002                                        X         PROC      1
000003                                        MOVE*     NAME
000004                                                  DO  X(1,*1) , LU  X(1,1)
000005                                                  DO  X(1,*2) , LL  X(1,2)
000006                                                  DO  X(1,*3) , GO  X1
000007                                                  BT  X(1,3)
000008                                                  DO  1        , END
000009                                        X1        NAME
000010                                                  DO  X(1,3)<04000 , LBK  X(1,3)
000011                                                  DO  X(1,3)>03777 , LB   (X(1,3))
000012                                                  SLJI      (MOVSUB)
000013                                                  END
000014                                        *
000015                                        * ***     CALL NAME
000016                                        *
000017          00 000000    10 0367                    MOVE      *(FROM),*(TO),12
                00 000001    12 0370
                00 000002    5070 14
000018          00 000003    10 0367                    MOVE      *(FROM),*(TO),*120
                00 000004    12 0370
                00 000005    36 0170
       U        00 000006    30 0371
000019          00 000007                     FROM      RES       120
000020          00 000177                     TO        RES       120
000021                                                  END
                00 000367    000007
                00 000370    000177
       U        00 000371    000000
                                                   *** SUMMARY ***

   PROGRAM SIZE:   00 00372

   EXTERNAL OR UNDEFINED REFERENCES:   MOVSUB
```

Explanation:

- Lines 2 and 13 define the procedure X.
- Line 3 provides the entry point MOVE.
- Lines 4 and 5 generate LU and LL if the first two parameters are preceded by an asterisk.
- The GO directive on line 6 will be performed if the third parameter is preceded by an asterisk. If so, lines 7 and 8 are ignored and the procedure subassembly resumes at line 9.
- If no asterisk appears in the third parameter, line 7 generates a BT instruction.
- Line 8 terminates the subassembly of the procedure. Note that the DO statement is used to avoid the termination of the procedure sample which would result if just an END statement were coded.
- Lines 10 and 11 generate either LBK or LB, depending on the number of words to be transferred.
- Line 12 generates an SLJI call on the subroutine MOVSUB.
- Line 17 calls on the procedure X through the MOVE entry. Since no asterisk precedes the third parameter, a BT 12 instruction is generated in addition to the LU and LL instructions.
- Line 18 calls on the same procedure but because the third parameter is preceded by an asterisk, the code generated is:

```
LU   (FROM)
LL   (TO)
LBK  120
SLJI (MOVSUB)
```

Example 2:

```
000001                                          /*
000002                                          X          PROC       1
000003                                          MOVE*      NAME
000004                                                     DO  X(1,*1) , LU  X(1,1)
000005                                                     DO  X(1,*2) , LL  X(1,2)
000006                                                     DO  X(1,*3) , GO  X1
000007                                                     BT  X(1,3)
000008                                                     END
000009                                          X1         PROC       0,2
000010                                                     DO  X(1,3)<04000 , LBK  X(1,3)
000011                                                     DO  X(1,3)>03777 , LB   (X(1,3))
000012                                                     SLJI       (MOVSUB)
000013                                                     END
000014                                          .
000015                                          . ***      CALL NAME
000016                                          .
000017            00 000000   10 0367                      MOVE       *(FROM),*(TO),12
                  00 000001   12 0370
                  00 000002   5070 14
000018            00 000003   10 0367                      MOVE       *(FROM),*(TO),*120
                  00 000004   12 0370
                  00 000005   36 0170
          U       00 000006   30 0371
000019            00 000007                     FROM       RES        120
000020            00 000177                     TO         RES        120
000021                                                     END
                  00 000367   000007
                  00 000370   000177
          U       00 000371   000000
                                                      *** SUMMARY ***

   PROGRAM SIZE:    00 00372

   EXTERNAL OR UNDEFINED REFERENCES:    MOVSUB
```

Explanation:

Example 1 is functionally identical to example 2. Instead of a single procedure, two separate procedures, X and X1, are defined, and the GO directive is used to transfer into the second procedure.

- If the NAME directive referred to in the GO statement is not defined or is not accessible at the level of subassembly of the GO directive, an expression error results (E flag) and scanning of the procedure resumes at the next line of the procedure sample.

- The GO directive may direct the assembler to resume processing of the subassembly at the occurrence of the specified NAME directive. The NAME directive may appear anywhere; that is, it may be a forward or back reference, or it may be a transfer into another procedure. As a result, great care must be taken to avoid infinite loops, caused by using the GO directive inappropriately.

3.12.3. DO Directive

The DO directive, as previously explained, is used to conditionally generate one or more words of data. The DO directive in the assembler is a powerful tool which, when used within procedures, provides great flexibility and power. When combined with the GO directive, the DO directive can be used to generate series of instructions iteratively as well as conditionally. The following paragraphs detail the rules which apply when these two directives are used together.

3.12.3.1. Conditional DO

If one of the conditional operators, < = or > , govern the determinant expression in the DO, or if the determinate expression has a value of 1, the GO directive is performed exactly as though the DO directive were absent. Therefore, the expressions:

DO 1 , GO A

and

GO A

are functionally identical.

3.12.3.2. Generative DO

If the determinant expression of the DO directive is greater than 1, the DO is said to be of the generative type. When the GO directive appears as the DO-item of a generative DO, the GO is performed iteratively as many times as the repeat count specifies. When an END directive is encountered, the next GO is performed. When the DO count is exhausted, processing continues at the statement following the DO.

Example:

```
000001                                          /.
000002                                          X          PROC
000003                                          LDSTOR*    NAME
000004                                          I          DO X  , GO X1
000005                                                     DO 1  , END
000006                                          X1         NAME
000007                                                     LA         X(I,1)
000008                                                     SA         X(I,2)
000009                                                     END
000010                                          .
000011                                          . ***      CALL NAME
000012                                          .
000013       00 000000     10 0014                         LDSTOR     A,B C,D E,F
             00 000001     12 0015
             00 000002     46 0016
             00 000003     44 0017
             00 000004     10 0020
             00 000005     12 0021
             00 000006     46 0022
             00 000007     44 0023
             00 000010     10 0024
             00 000011     12 0025
             00 000012     46 0026
             00 000013     44 0027
000014                                          .
000015       00 000014     201400               A          +1.0
             00 000015     000000
000016       00 000016                          B          RES        2
000017       00 000020     202600               C          +3.0
             00 000021     000000
000018       00 000022                          D          RES        2
000019       00 000024     203500               E          +5.0
             00 000025     000000
000020       00 000026                          F          RES        2
000021                                                     END
```

Explanation:

- Lines 2 and 9 define the procedure X.
- Line 3 provides the entry point LDSTOR.

- Line 4 combines a DO and GO directive. Since the paraform X may have a value between 0 and infinity (actually the maximum number of fields allowed is 176_8), it may be either a conditional or generative DO. Assuming X>1, the DO is of the generative type. As a result the GO X1 is performed X times. Each time, transfer is made to line 6, and the procedures LA and SA are performed. After the DO count is exhausted, line 5, which terminates the subassembly, is performed. If X=1, transfer to line 6 is made, and subassembly is terminated upon encountering the END directive in line 9.

- Line 7 calls the procedure LA, which generates the instructions LU and LL.

- Line 8 calls the procedure SA,which generates an SU and SL.

- Line 13 calls on the procedure X and generates the instructions:

```
LU   A
LL   A+1
SU   B
SL   B+1
LU   C
LL   C+1
SU   D
```

  and so on through F.

Example:

```
000001                                                   /*
000002                                                   X        PROC
000003                                                   CALLIO*  NAME
000004                                                            SLJI       (OPEN)
000005                                                   I        DO   X+2 , GO X1
000006                                                            SLJI       (CLOSE)
000007                                                   X1       NAME
000008                                                            DO   I>X , END
000009                                                            LLK        X(I,1)
000010                                                            SLJI       (GET)
000011                                                            END
000012                                                   .
000013                                                   . ***    CALL NAME
000014                                                   .
000015    U      00 000000    30 0014                             CALLIO  1 3 5
                                + 6,12 00
                 00 000001    70 0001
                                + 6,12 00
          U      00 000002    30 0015
                                + 6,12 00
                 00 000003    70 0003
                                + 6,12 00
          U      00 000004    30 0015
                                + 6,12 00
                 00 000005    70 0005
                                + 6,12 00
          U      00 000006    30 0015
                                + 6,12 00
          U      00 000007    30 0016
                                + 6,12 00
000016    U      00 000010    30 0014                             CALLIO  2
                                + 6,12 00
                 00 000011    70 0002
                                + 6,12 00
          U      00 000012    30 0015
                                + 6,12 00
          U      00 000013    30 0016
                                + 6,12 00
000017                                                            END
          U      00 000014    000000
                                +    18 00     OPEN
          U      00 000015    000000
                                +    18 00     GET
          U      00 000016    000000
                                +    18 00     CLOSE

                                                         *** SUMMARY ***

   PROGRAM SIZE:   00 00017

   EXTERNAL OR UNDEFINED REFERENCES:     CLOSE     GET      OPEN
```

Explanation :

- Lines 2 and 11 define the procedure X.
- Line 3 provides the entry point CALLIO.
- Line 4 generates a call to subroutine OPEN.
- Line 5 is a generative DO directive which transfers to line 7. The determinant value is forced to be greater than 1 so that line 6 must always be generated upon completion of the DO. For each parameter supplied, an LLK parameter value and an SLJI (GET) are generated at lines 9 and 10.
- Line 6 generates a call to subroutine CLOSE.
- Line 8 terminates both the DO count when I>X and the subassembly after the generation of line 6.

- Line 15 calls on the procedure through entry point CALLIO to generate the instructions:

```
SLJI   (OPEN)
LLK     1
SLJI   (GET)
LLK     3
SLJI   (GET)
LLK     5
SLJI   (GET)
SLJI   (CLOSE)
```

- Line 16 calls on the procedure through entry point CALLIO to generate the instructions:

```
SLJI   (OPEN)
LLK    2
SLJI   (GET)
SLJI   (CLOSE)
```

4. ASSEMBLER OPERATION

4.1. GENERAL

This section discusses the ways in which the assembler is to be used, what results are produced, and the meaning of the error diagnostics and messages which may result during the operation of the assembler.

4.2. CONTROL CARD FORMAT

The assembler is an element of the Real Time Operating System (RTOS) and operates under its control. The assembler may be called upon to assemble a symbolic program through the use of the @ASM control card.

The @ASM control card has the form:

```
@ASM,options   pronam
```

The program name, designated by the parameter pronam, is the name of the symbolic element to be assembled, and will be the name given to the produced relocatable object code element.

If no options are to be exercised, the comma following the ASM function may be omitted. At least one blank character must follow the option field. If no options are specified, the symbolic statements to be assembled must immediately follow the control card. Upon the occurrence of either another control card or an END directive which does not signify the end of a procedure sample, the assembler is terminated.

The following options may be present on the @ASM control card:

T – Results in listing all inserted elements.

M – Results in listing the mode value of all data words generated.

N – Results in the omission of all listings except those statements containing an error flag.

A – Results in the omission of all listings.

R – Results in the listing of a cross-reference of all labels referenced in the assembly after the assembly is complete.

P – Results in the punching of a relocatable object-code card element.

* – Specifies that the source to be assembled is to be found in the user run library as a symbolic element. Correction cards may follow the @ASM control card.

NOTE: An A option overrides the presence of the N, T, and R options.

4.3. ASSEMBLER OUTPUT LISTING

Unless an A or N option is present on the @ASM control card, the assembler produces a printed listing of the symbolic statements processed together with the code produced.

Example:

```
00:01:34        @ASM,*T  T4-1

                UNIVAC  418-III ASSEMBLY --   MAR 17 1970    00:01:34
000001                                             START,
000002     U       00 000000    30 0015                         SLJI      (FRMBUF)
000003                                                          INSERT    LEVEL1
000001 01 D        00 000001    12 0057                         LL        XX2
000002 01          00 000002    10 0001                         LU        XX3
000003 01          00 000003    32 0002                         LB        XX4
000004 01                                                       INSERT    LEVEL2
000001 02          00 000004    76 0007                         SLJ       PRINT1
000002 02          00 000005    32 0001                         LB        XX3
000003 02          00 000006    34 0000                         J         START
000004 02                                                       INSERT    LEVEL3
000001 03          00 000007    000000             PRINT1*      +         0
000002 03 E        00 000010    001004                          PRINTS    PR1  BUF,44,1
                   00 000011    000014
                   00 000012    000003
                   00 000013    000454
000003 03          00 000014    55 0007            PR1          JI        PRINT1
000004 03                                                       INSERT    LEVEL4
000001 04                                          S(10),
000002 04 D        10 000000    000000             XX2          +         0
000003 04          10 000001    000000             XX3          +         0
000004 04          10 000002    000000             XX4          +         0
000005 04          10 000003                       BUF          RES       44
000004    D        10 000057    000000             XX2          +         0
000005                          000000                          END       START,2
          U        00 000015    000000

                                                            *** SUMMARY ***

  PROGRAM SIZE:    00 00016     10 00060

  EXTERNAL OR UNDEFINED REFERENCES:   FRMBUF

  EXTERNAL DEFINITIONS:               PRINT1

  DOUBLY DEFINED LABELS:              XX2

  EXPRESSION ERRORS:   001

  INSERTED ELEMENTS:   LEVEL1  BY  T4-1
                       LEVEL2  BY  LEVEL1
                       LEVEL3  BY  LEVEL2
                       LEVEL4  BY  LEVEL3
```

Explanation:

- Field 1 contains the line number of the symbolic statement.
- Field 2 is present only when the symbolic code being assembled comes from an inserted element, and identifies the level of inserted elements.
- Field 3 is present only if diagnostic warnings are produced, and identifies the type of error detected.
- Field 4 identifies the active location counter.
- Field 5 contains the relative value of the active location counter.
- Fields 6 and 7 contain the binary value of the code generated.

- The remainder of the line reflects the supplied symbolic image.
- Following the END directive, all literals are printed.

The summary printed at the conclusion of the assembly specifies:

- the size of each location counter used;
- the names of external or undefined labels;
- the names of any externally defined labels;
- the names of any doubly defined labels;
- the number of diagnostics that occurred during the assembly;
- the names of any inserted elements and the elements which caused their insertion.

4.3.1 Mode Listing

If the M option is present on the @ASM control card, each line of generated code is followed by the mode value of the data word produced. The mode value indicates:

- the size of the relocation field;
- the location counter of the operand field;
- the label of an external reference;
- the presence of the IBOO operator;
- whether the data word is to be relocated;
- whether positive or negative relocation is specified;
- whether the external reference is to be added or subtracted.

The format of the mode value line listed is:

```
        I  r  fs  lc  s   label
```

where:

I	is present if the IBOO operator is present in the expression.
r	is + if positive relocation is specified; is – if negative relocation is specified; is blank if no relocation is to be performed.
fs	is 18 if the entire data word may be relocated or modified by the value of the external reference; is 6,12 if the lower 12 bits may be relocated or modified by the value of the external reference.

lc is the location counter under which the operand expression is to be relocated, and is 0 if the operand field in nonrelocatable.

s is blank if the value of the external reference is to be added;

is – if the value of the external reference is to be subtracted.

label is the name of the external reference.

NOTE: The M option should be used only if there is a need to examine the mode values generated. Even though the assembly is not significantly slowed down, an extra line of print is generated for each word and may cause an early overflow of the symbiont drum space.

Example:

```
00:01:35            @ASM,M     T4-2

                    UNIVAC  418-III ASSEMBLY --   MAR 17 1970     00:01:35
000001                 00 000000                                  RES         8
000002                 00 000010    70 0001                 TAG   LLK         1
                                      + 6,12 00
000003                 00 000011    12 0010                       LL          TAG
                                      + 6,12 00
000004                 05 000000    000000            $(5),TAG5   +0
                                      +    18 00
000005                 00 000012    12 7777                 $(0)  LL          -TAG5
                                      - 6,12 05
000006                 00 000013    000010                        +TAG
                                      +    18 00
000007                 00 000014    777777                        -TAG5
                                      -    18 05
000008                 00 000015    12 0021                       LL          (TAG5!)
                                      + 6,12 00
000009      U          00 000016    12 0003                       LL          UREF+3
                                      + 6,12 00    UREF
000010      U          00 000017    12 0006                       LL          3*2-UREF+TAG5
                                      + 6,12 05  - UREF
000011      U          00 000020    000000                        -UREF!
                                     I+    18 00  - UREF
000012                                                            END
                       00 000021    000000
                                     I+    18 05
                                                             *** SUMMARY ***

      PROGRAM SIZE:   00 00022    05 00001

      EXTERNAL OR UNDEFINED REFERENCES:   UREF
```

Explanation:

- Line 2 is a constant.
- Line 3 is 12-bit relocatable. The operand value is to be relocated under location counter 0.
- Line 4 is a constant.
- Line 5 is 12-bit relocatable. The 12-bit operand value is to be relocated under location counter 5. Relocation is negative; that is, the relocation base is to be subtracted.
- Lines 6 and 7 are 18-bit relocatable.
- Line 8 itself is 12-bit relocatable. The referenced literal is 18-bit relocatable and IBOOed.

- Line 9 is 12-bit modified by the value UREF.
- Line 10 is 12-bit relocatable by location counter 5, and 12-bit modified by the value −UREF.
- Line 11 is 18-bit modified by the value −UREF and IBOOed.

4.3.2. Cross-Reference Listing

If an R option is present on the @ASM control card, a cross-reference listing of all referenced labels will be produced at the end of the assembly. Although the cross-reference itself does not significantly slow down the assembly, six words of storage are used for each label reference when the R option is present. As a result the assembler label table space requirements may be significantly larger during the assembly with an R option.

At the conclusion of the assembly, all referenced labels are printed in alphabetic order together with the location counter value at which they were assigned, the location under which they are defined, and the subassembly level at which they were defined. The location counter and location counter value of each reference to the label are also printed.

If a reference is made to a labeled constant, the decimal value of the constant is printed. The octal value is printed between brackets.

Example:

```
00:01:36        @ASM,R     T4-3

                UNIVAC  418-III ASSEMBLY --   MAR 17 1970    00:01:36
000001                                         X*       PROC
000002                                         I        EQU       10
000003                                         *        LLK       I
000004                                                  END
000005             00 000000   12 0005         START    LL        LB1
000006      U      00 000001   70 0000                  LLK       UND
000007      U      00 000002   5070 00                  BT        I
000008             00 000003   12 0006                  LL        LB2
000009             00 000004   12 0005                  LL        LB1
000010             00 000005   70 0012         LB1      X
000011             00 000006   70 0012         LB2      X
000012             00 000007   34 0000                  J         START
000013                                                  END
                                                    *** SUMMARY ***

   PROGRAM SIZE:    00 00010

   EXTERNAL OR UNDEFINED REFERENCES:    I        UND
                                                 *** CROSS REFERENCE LISTING ***

I       ,UNDEFINED              LEVEL 00: REFERENCED AT LINE(S):  - 00002 00
I       = 000010  (000012)      LEVEL 01: REFERENCED AT LINE(S):  - 00005 00   - 00006 00
LB1     DEFINED AT 000005 00 LEVEL 00: REFERENCED AT LINE(S):  - 00000 00   - 00004 00
LB2     DEFINED AT 000006 00 LEVEL 00: REFERENCED AT LINE(S):  - 00003 00
START   DEFINED AT 000000 00 LEVEL 00: REFERENCED AT LINE(S):  - 00007 00
UND     ,UNDEFINED              LEVEL 00: REFERENCED AT LINE(S):  - 00001 00
```

4.4. SYMBOLIC CORRECTIONS

If the * option is present, the symbolic code is assembled from the user run library. Corrections may be made to the symbolic code. Correction cards immediately follow the @ASM control card and are terminated by the occurrence of another control card.

The line numbers listed in the first column of the assembly are used to indicate whicl images are to be removed or altered. Correction cards are not added to the symbolic element in the library. Correction cards do not cause the line numbers on the listing to be changed, so that no matter how many corrections are made, the line numbers still reflect those associated with the original symbolic element.

Symbolic lines which are deleted as a result of the supplied corrections are marked and listed with --- following the line number. They are not assembled.

New symbolic images supplied in the correction deck are marked with +++ following the line number. The line number associated with new symbolic images is that of the last statement in the original element.

Example:

```
00:01:36          @ASM,*     T4-4

                  UNIVAC  418-III ASSEMBLY --   MAR 17 1970     00:01:36
000001---                                       ---    START     LLK         1
000001+++           00 000000     12 0046              START     LL          (TO)
000002              00 000001     10 0047                        LU          (FROM)
000003              00 000002     5070 07                        BT          7
000004---                                       ---              MOVE        FROM,(TO)
000004+++           00 000003     10 0047                        MOVE        FROM,TO,7
                    00 000004     12 0046
                    00 000005     5070 07
000005              00 000006     770300                         EOJ$
000006              00 000007                          FROM      RES         25
000006+++           00 000040     663334               TO        'THIS IS DESTROYED'
                    00 000041     650034
                    00 000042     650027
                    00 000043     306566
                    00 000044     545173
                    00 000045     302700
000006+++                         000000                         END         START,3
                    00 000046     000040
                    00 000047     000007
```

Correction cards having the following PUR-compatible format. Lines are deleted by specifying:

−n,m

where n is the first and m is the last line to be deleted. Following the correction card, symbolic statements may be supplied. These are inserted in place of the deleted images.

In order to add new symbolic images, the card:

−n

where n is the line following which corrections are to be inserted, is supplied. The symbolic statements to be added after line n follow the correction card.

Correction cards must be supplied in ascending line number sequence. The − must occur in column 1 of the correction card.

Example:

```
   LABEL            OPERATION                            OPERAND
1          10                   20                   30                   40
-1,1
START     LL        (TO)
-4,4
          MOVE      FROM,TO,7
-6
TO        'THIS IS DESTROYED'
          END       START,3
```

4.5. DIAGNOSTICS

Errors detected by the assembler in processing a symbolic statement are flagged. Depending on the particular error, the code may or may not be generated correctly. Some diagnostic flags are not indicative of errors but are warnings.

4.5.1. Address Warning (A)

The address warning diagnostic A is generated if the 12-bit operand address of a type I or II instruction has a location counter value such that the instruction and the referenced address are in different bays. If the operand address is relocated under a different location counter from the instruction, no A-flag will be generated.

Example:

```
000001                                          /.
000002     A       00 000000    12 0010                   LL        A
000003             00 000001                              RES       010007
000004             00 010010    000000          A         +0
000005             00 010011    12 0007                   LL        B
000006             01 000000                    S(1)      RES       010007
000007             01 010007    000000          B         +0
000008                                                    END
                                                    *** SUMMARY ***

   PROGRAM SIZE:   00 10012     01 10010

   ADDRESS WARNINGS:    001
```

4.5.2. Format Warning (F)

The format warning is generated if a 2,16 FORM directive has a relocatable address reference in the second expression.

Example:

```
000001                                         /*
000002                                         IF       FORM     2,16
000003   F      00 000000     000001                    IF       0,LABEL
000004          00 000001     70 0001          LABEL    LLK      1
000005                                                  END
```

4.5.3. Truncation Warning (T)

A field truncation warning is generated if:

- the value of an expression in a FORM reference exceeds the size of the field; or
- the value of the operand field in a type I or II instruction is a constant exceeding 07777. If the constant is negative and the instruction is an LBK, LLK, or ALK, the T flag is not generated.

Example:

```
000001                                         /*
000002                                         IF       FORM     5,4,5,4
000003   T      00 000000     160064                    IF       7,2000,3,20
000004          00 000001     70 7772                   LLK      -5
000005   T      00 000002     70 0000                   LLK      010000
000006          00 000003     12 3720                   LL       2000
000007   T      00 000004     12 1610                   LL       5000
000008   T      00 000005     64 7765                   JUP      -10
000009                                                  END
                                                   *** SUMMARY ***

    PROGRAM SIZE:     00 00006

    TRUNCATION WARNINGS: 004
```

4.5.4. Level Error (EL)

A level error indicates that the number of nested expressions exceeded the maximum of six. The resulting expression value is 0.

Example:

```
000001                                      /*
000002    EL     00 000000     12 0003                                       LL     (1+(1+(1+(1+(1+(1+(1+2)))
000003                         000002       A(1)                             EQU    2
000004    EL     00 000001     000000       A(A(A(A(A(A(A(1)))))))))         EQU    2
000005           00 000002     000001                                        +A(1)
000006                                                                       END
                 00 000003     000007
                                                  *** SUMMARY ***

    PROGRAM SIZE:      00 00004

    EXPRESSION ERRORS:     002

    LEVEL ERRORS:          002
```

4.5.5. Instruction Error (I)

An instruction error indicates that the assembler detected an illegal operation field or label field specification. An I flag is generated if:

- a symbolic label is detected in the operation field which is not a defined label, procedure entry point, assembler directive, FORM name, or mnemonic instruction;
- a location counter is defined in the label field and the terminating character is not a space, comma, or period;
- the label field is not terminated by a space, asterisk, or period;
- the type number on an INFO directive exceeds 7;
- the type field on an INFO directive is not terminated by a space;
- the location counter specified on an INFO directive exceeds 15 or is not terminated by a space or comma;
- an EQU directive does not have a label in the label field; or
- a procedure call line references a procedure entry point in the parameter expressions.

Example:

```
00:01:40        @ASM      T4-10

                UNIVAC  418-III ASSEMBLY --   MAR 17 1970    00:01:40
**PROCEDURE JZL    NOT IN LIBRARY - CALLED AT LINE 000001    BY ELEMENT T4-10
000001    UEI     00 000000    000000                          JZL    $+1
000002    I       10 000000    34 0001           $(10)LABEL    J      $+1
000003    I                                      LABEL+0
000004    I                                                    INFO   8    0
000005    I                                                    INFO   7,0
000006    I                                                    INFO   7    17
000007    I                    000005            $(1)          EQU    5
000008                                           X*            PROC   1,1
000009                                                         +5
000010                                                         END
000011    I       01 000000    000005                      X          (X )
000012                                                         END

                                                    *** SUMMARY ***

   PROGRAM SIZE:    00 00001    01 00001    10 00001

   EXTERNAL OR UNDEFINED REFERENCES:   JZL

   EXPRESSION ERRORS:   001

   INSTRUCTION ERRORS:  008
```

4.5.6. Relocation Error (R)

Relocation warnings or errors are generated if elementary items are combined in such a way as to cast doubt on the validity of the expression.

Relocation errors are generated if a relocatable item is combined with a constant. See Table 1–3 for details of allowed mode combinations.

Example:

```
000001                                               /*
000002            00 000000                                      RES       5
000003    R       00 000005   000005                 A           +A**5
000004            00 000006   000240                             +5*/A
000005    R       00 000007   000003                             +A//2
000006            00 000010   000005                             +A*1
000007    R       00 000011   000012                             +A*2
000008                                                           END

                                                         *** SUMMARY ***

     PROGRAM SIZE:    00 00012

     RELOCATION WARNINGS: 003
```

4.5.7. External or Undefined Warning (U)

The U flag is set when a label is referenced which is not defined in the assembly. If the label is externally defined in some other element, the loader collects the elements.

Example:

```
000001                                               /*
000002    U       00 000000   70 0000                            LLK       ABC
000003    U       00 000001   12 0002                            LL        (LABEL)
000004                                                           END
          U       00 000002   000000

                                                         *** SUMMARY ***

     PROGRAM SIZE:    00 00003

     EXTERNAL OR UNDEFINED REFERENCES:    LABEL     ABC
```

4.5.8. Double Definition Warning (D)

A double definition warning is generated when the value assigned to a label changes. The assembler processes the symbolic code twice. As a result, a D flag may indicate that a label is defined at different relative locations because of a pass conflict; that is, different amounts of code were generated in pass 1 and pass 2 of the assembly. The D flag is set if:

- a label defined previously is redefined to have a different value. If the label is a dimensioned label, the D flag is suppressed;
- a label defined in pass 1 of the assembly does not have the same value when redefined in pass 2;
- a paraform contains a reference to a doubly defined label;

- a PROC or NAME entry point is defined or multiply defined, regardless of the level of the entry point;
- a literal contains a reference to a doubly defined line item; or
- an expression references a doubly defined label.

Example:

```
000001                                                /*
000002    D      00 000000    12 0002                 TA        LL        TB
000003    D      00 000001    12 0000                 TB        LL        $-1
000004    D      00 000002    000000                  TB        +0
000005    D                   000001                  A         EQU       1
000006    D                   000002                  A         EQU       2
000007                                                P*        PROC
000008                                                          +0
000009                                                          END
000010           00 000003    000000                            P
000011                                                P         PROC
000012                                                          +1
000013                                                          END
000014           00 000004    000000                            P
000015    D      00 000005    12 0006                           LL        (TB)
000016                                                          END
                 00 000006    000002

                                                           *** SUMMARY ***

  PROGRAM SIZE:    00 00007

  DOUBLY DEFINED LABELS:              A        P        P        TB
```

4.5.9. Expession Errors (E)

Expression errors indicate that the syntax rules for defining an expression were not obeyed. Expression errors are generated if:

- an elementary item is not followed by an operator or terminator;
- an operator is not followed by an elementary item;
- a floating-point number is written so as to have an octal integer part;
- a floating-point number exceeds the maximum value;
- an item is multiplied by, divided by, or compared with an item which is undefined;
- a paraform with subscript is not terminated by a comma or bracket;
- two items are compared which do not have the same mode value;
- two items are compared and one is undefined;
- two relocatable items are combined and are relocated under different location counters;
- an alphastring or double floating-point number occurs in a literal;
- a dimensioned label is defined or referenced which has previously been defined with smaller dimensionality;
- an LSD or SSD instruction is indexed;

- a type II instruction is indexed;
- a FORM reference has more than the allowed number of operand expressions;
- a GO directive is used and no operand expression is present;
- a FORM directive is defined for more than the allowed field sizes;
- an INSERT directive is specified without operand field;
- a label has more than six characters; or
- a location counter larger than 15 is referenced or defined.

Example:

```
000001                                      /•
000002   E     00 000000   000005                     +5X
000003   E     00 000001   000000           A         +5+
000004   E     00 000002   204406                     +010•2
               00 000003   314631
000005   E     00 000004   000000                     +10•E50
               00 000005   000000
000006   E     00 000006   000000                     +3*ULBL
000007                                      X*        PROC
000008                                                +2
000009                                                +X(1,
000010                                                GO
000011                                                +3
000012                                                END
000013         00 000007   000002                     X         1
         E     00 000010   000000
         UE    00 000011   000003
000014   E                                            DO        A>2 , +3
000015   E     00 000012   000000                     +ULBL>2
000016   E     00 000013   000001                     +A+B
000017         05 000000                    $(5),B    RES       5
000018   E     00 000014   12 0033          $(0)      LL        ('ABCDEFGH')
000019   E     00 000015   12 0035                    LL        (1•D3)
000020         00 000016   70 0005          D(1,1)    LLK       5
000021   E     00 000017   000000           D(1,1,1)  LLK       3
000022   E     00 000020   12 0000                    LL        D(1,1,1)
000023   E     00 000021   502000                     LSD       *A
               00 000022   000001
000024   E     00 000023   64 0001                    JUP       *A
000025                                      IFA       FORM      3,15
000026   E     00 000024   100002                     IFA       1,2,3
000027   E                                  IF        FORM      2,15,3
000028   E                                            INSERT
000029   EI                                 LABEL7C   LLK       3
000030   E     00 000025   12 0026                    LL        $(17)+1
000031   E     00 000026                    $(17)     RES       5
000032                                                END
               00 000033   242526
               00 000034   273031
               00 000035   000000
               00 000036   000000

                                                 *** SUMMARY ***

   PROGRAM SIZE:   00 00037    05 00005

   EXTERNAL OR UNDEFINED REFERENCES:   ULBL

   EXPRESSION ERRORS:   022

   INSTRUCTION ERRORS:   001
```

4.6. ERROR MESSAGES

When abnormal situations arise in the course of an assembly, the assembler prints a message which specifies what happened and continues or terminates depending on the nature of the problem.

4.6.1. Element Not Found

If a symbolic element is to be inserted and cannot be found in either the user or system library, the message:

```
*** ELEMENT xxxxxx NOT IN LIBRARY , CALLED AT LINE 111111 BY ELEMENT
    cccccc
```

is printed and the INSERT directive is ignored.

4.6.2. Procedure Not Found

If a procedure is referenced which is not defined in the program and is not present in the user or system library, the message:

```
*** PROCEDURE xxxxxx NOT IN LIBRARY , CALLED AT LINE 111111 BY
    ELEMENT cccccc
```

is printed and the procedure call is assumed to be a label reference. Note that a possible procedure call is signified by the occurrence of a symbolic label in the operation field which is not a previously defined FORM reference or mnemonic instruction.

4.6.3. END Card Omission

If the symbolic statements are not terminated by an END directive, the assembler inserts the image:

```
END  *** ART GENERATED ***
```

4.6.4. Drum Library Overflow

If the code generated in the course of the assembly causes the library to overflow, the message:

```
***ASSEMBLY ABORTED – DRUM LIBRARY OVERFLOW***
```

is printed. The element is not placed in the library.

4.6.5. Main Storage Overflow

If the assembler attempts to obtain additional main storage space because the procedure sample storage or label table is filled, and no space is obtained, the assembly is terminated with the message:

```
***ASSEMBLY ABORTED – PROCEDURE TABLE OVERFLOW***
```

or

```
***ASSEMBLY ABORTED – LABEL TABLE OVERFLOW***
```

Prior to terminating, the assembler tries to obtain as little as 512 words of memory to expand its tables.

4.6.6. Internal Error

If an error condition occurs within the assembler, the message:

ASSEMBLY ABORTED – INTERNAL ERROR

is printed and the assembler terminates automatically. The programmer should try the run again. If the problem continues to occur, a report should be filed.

4.6.7. Element Deletion

At the conclusion of the assembly, the code produced is registered in the user library as a relocatable element. If a relocatable element by the same name already exists in the library, it is deleted and the message:

```
*** THE RELOCATABLE ELEMENT xxxxxx , (CREATED mm:dd:yy) ,
    HAS BEEN DELETED ***
```

is printed. The month (mm), day (dd), and year (yy) refer to the date that the deleted element was created.

4.6.8. Correction Errors

When correction cards are submitted, several errors may be detected.

If a correction card references a line number beyond the range of the element, the message:

*** LAST CORRECTION EXTENDS BEYOND ELEMENT ***

is printed, and the correction cards are ignored.

If a correction card references a line number smaller than one previously referenced, the message:

*** SEQUENCE ERROR ***

is printed and the correction card is listed and ignored.

If a correction card of the type –n,m is such that m<n, the message:

*** LINE NO. DESCENDING ***

is printed and the correction card is listed and ignored.

4.7. GENERATION PARAMETERS

When the RTOS system is generated, parameters may be supplied for the assembler. The assembler parameters are specified in the element CONFIG on a procedure call of the type:

```
ART,m    mlc    prs    lts
```

where:

mlc	is the maximum allowed value for any one location counter;
prs	is the reserved procedure table size;
lts	is the number of modules (6 words) reserved for the label table.
m	is the size of procedure or label table expansion in 256-word blocks. If left blank, m is assumed to be 16 (4096 words).

The assumed (supplied) parameters are:

```
ART  030000  300  100  .
```

Procedure or label table space is expanded as needed in modules of 256*m maximum words until no space is available. The maximum location counter value is used to detect program-directed assembly loops (GO directive which does not terminate).

In order to change the assembler generation parameters the symbolic element ARTGEN must be assembled with the appropriate CONFIG element.

4.8. ELEMENT AND PROCEDURE INSERTION

The INSERT directive causes a symbolic element in the library to be included as part of the assembly. A procedure reference to a procedure entry point not defined in the program may cause the procedure sample to be inserted from the library.

First, the user RUN library is searched for the element or entry point. If not found, the system library is searched.

If a procedure entry point is referenced, the entire procedure element, which may include other procedure samples, is brought into procedure storage. As a result, care should be taken to ensure that a procedure reference does not cause another procedure in the same element to be read into storage which has entry points which duplicate already defined procedure entry points.

4.9. LABEL TABLE REFERENCES

Symbolic items are stored in the label table. Whe n a reference is made to a symbolic item, the label table is searched. If the same symbolic label is used for different types of symbolic items, the first acceptable definition for the label is used. The first acceptable reference is determined by the sequence in which the assembler searches the label table. The sequence is defined in the following paragraphs.

4.9.1. Operand Field Hierarchy

A symbolic item referenced in the operand field may be a label item, a paraform reference, a dimensioned label item, or a labeled literal reference. The sequence in which the assembler searches the label table is:

1. label item
2. paraform item
3. dimensioned label item
4. labeled literal

Example:

```
000001                                              /.
000002              00 000000                                   RES       10
000003                                              A           LIT
000004              00 000012    000013                         +A(LLK 10)
000005                                              $(1),B      LIT
000006              01 000000                       B(1)        RES       10
000007              01 000012    000000                         +B(1)
000008                                              $(2),C      LIT
000009                                              C*          PROC
000010                                                          +C(1)
000011                                                          +C(1,1)
000012                                                          +C(1,2)
000013                                                          +C
000014                                                          END
000015                           000144             C(1)        EQU       100
000016              02 000000    000001                         C         C(1)  C
                    02 000001    000144
                    02 000002    000000
                    02 000003    000002
000017                                              $(3),D      LIT
000018              03 000000                                   RES       10
000019              03 000012    70 0012            D           LLK       D
000020                           000144             D(1)        EQU       100
000021      UE      03 000013    12 0000                        LL        D(LLK 0)
000022                                                          END
                    00 000013    70 0012
                                                         *** SUMMARY ***

  PROGRAM SIZE:   00 00014     01 00013     02 00004     03 00014

  EXTERNAL OR UNDEFINED REFERENCES:   LLK

  EXPRESSION ERRORS:   001
```

4.9.2. Operation Field Hierarchy

A symbolic label occurring in the operation field may be a procedure entry point, a directive reference, an instruction reference, or a label reference. The assembler determines the nature of the label as follows:

1. If the field is terminated by a space (blank character), a check is made for an INFO, LIT, NAME, PROC, FORM, EQU, DO, XCHAR, UNLIST, EVEN, ODD, GO, GO, RES, END, LIST, INSERT, SKIP, or CHAR directive.

2. If the label is not a directive or if the field terminator is a comma, a check is made for a procedure entry point.

3. If the field terminator is not a space or commas, the field is assumed to be the operand field, and one or more data words are generated.

4. A check is made for a FORM reference.

5. A check is made for a mnemonic instruction reference.

6. A check is made for a library procedure entry point; and, if found, the procedure sample is brought into procedure storage.

7. If none of the forgoing references are satisfied, the field is scanned as an operand field expression.

The sequence described shows that:

1. A label with a name that is identical to an assembler directive may be used as a procedure entry point if and only if a comma is used to terminate the operation field.

2. A procedure entry point or form reference which has the same label as a mnemonic instruction will supercede the instruction reference unless the procedure entry point is only defined in the library and not yet brought into procedure storage.

3. A label reference not preceded by a + will cause the procedure library to be searched prior to assuming a data word generation format.

Example:

```
00:01:45        @ASM        T4-16

                UNIVAC  418-III ASSEMBLY --   MAR 17 1970    00:01:45
**PROCEDURE XCHAR  NOT IN LIBRARY - CALLED AT LINE 000010    BY ELEMENT T4-16
**PROCEDURE A      NOT IN LIBRARY - CALLED AT LINE 000012    BY ELEMENT T4-16
000001                                             /.
000002                                             LB        FORM      1,3,14
000003           00 000000   400024                          LB        1,0,20
000004                                             CHAR*     PROC
000005                                                       LLK       CHAR(1,1)
000006                                                       END
000007           00 000001   70 0024                         CHAR,0    'A'
000008                                                       CHAR      'A',6
000009           00 000002   70 0006                         CHAR,0    'A'
000010     UI    00 000003   000000                          XCHAR,0
000011           00 000004                         A         RES       5
000012           00 000011   000004                          A
000013                                                       END
                                                       *** SUMMARY ***

    PROGRAM SIZE:    00 00012

    EXTERNAL OR UNDEFINED REFERENCES:    XCHAR

    INSTRUCTION ERRORS:  001
```

Note that a literal contains a line item which begins with the operation field. As a result, there is a difference between the way that the references in the following example of

LL (A)

and

LL (A)

are treated because in the first literal, the operation field terminator precludes a reference to a procedure entry point.

Example:

```
00:01:46        @ASM      T4-17

                UNIVAC  418-III ASSEMBLY --   MAR 17 1970    00:01:46
**PROCEDURE A       NOT IN LIBRARY - CALLED AT LINE 000004    BY ELEMENT T4-17

000001                                            /*
000002              00 000000                                  RES       10
000003    U         00 000012   12 0014                        LL        (A)
000004    UI        00 000013   12 0014                        LL        (A )
000005                                                         END
          U         00 000014   000000
                                                     *** SUMMARY ***

   PROGRAM SIZE:   00 00015

   EXTERNAL OR UNDEFINED REFERENCES:   A

   INSTRUCTION ERRORS:  001
```

5. COMMAND/ARITHMETIC SECTION

5.1. GENERAL

In this section, the command/arithmetic section of the UNIVAC 418-III System is discussed. Since all input/output is normally done through executive requests, these hardware characteristics are not discussed in this document.

5.2. HARDWARE CHARACTERISTICS

The UNIVAC 418-III System may contain up to 131,072 addressable words. Each word consists of 18 bits. Main storage can be thought of as divided into 4096-word segments called bays.

The address of the instruction being executed is kept in a register called the instruction address register (IAR).

Eight index registers (B registers) can be used for address modification. The index registers are memory locations 1 through 10_8.

A 6-bit special register (SR) is used to access different bays. Four instructions are available to load and store the special register.

A 4-bit register called the index register pointer (IRP) contains the address of the active index register.

When abnormal conditions, such as illegal instructions, arithmetic overflow, or guard mode violation occur, the operating program is interrupted, and the instruction at a fixed (preassigned) address is executed.

5.3. DESIGNATORS

- Compare Designator

  The compare designator is a bi-stable, three-stage register whose state is determined by the execution of any of the COMPARE instructions (f = 02,03,06,07). The results of the COMPARE instructions are recorded by the compare designator as follows:

  – The COMPARE stage is set upon the execution of any of the COMPARE instructions.

  – The LESS THAN stage is set if a COMPARE instruction finds (AL) less than the contents of the addressed memory location (f = 02,03), or [(AU) **AND** (AL)] less than the logical product of (AU) and the contents of the addressed memory location (f = 06,07).

  – The EQUALS stage is set if a COMPARE instruction finds (AL) equal to the contents of the addressed memory location (f = 02,03) or [(AU) **AND** (AL)] equal to the logical product of (AU) and the contents of the addressed memory location (f = 06,07).

The COMPARE stage is cleared by the execution of any instruction other than the arithmetic JUMP instructions (f = 6067). Thus, if the results of a COMPARE instruction are to be successfully tested, it must be immediately followed by one or more of the JUMP instructions.

When the COMPARE stage of the compare designator is set, all interrupts are locked out to avoid the possibility of inadvertently clearing the COMPARE state. It should be noted that the arithmetic JUMP instructions have significantly different operations if executed when the COMPARE stage is not set.

- Borrow Designator

  The borrow designator is a bi-stable, single-stage element whose state is determined by the execution of either a double-length ADD instruction (f = 20,21) or a double-length SUBTRACT instruction (f = 22,23).

  If an end-around borrow is required during the execution of either of these instructions, the end-around borrow is inhibited and the borrow designator is set. The borrow designator remains set until the subsequent execution of another double-length ADD or double-length SUBTRACT instruction.

  The condition of the borrow designator may be tested by the TEST NO BORROW instruction (f = 5051). When the borrow designator is set, interrupts are not locked out.

- Overflow Designator

  The overflow designator is a bi-stable, single-stage element set when an overflow occurs during the execution of any of the following instructions:

  ADD AL (f = 14,15)
  SUBTRACT AL (f = 16,17)
  ADD A (f = 20,21)
  SUBTRACT A (f = 22,23)
  DIVIDE A (f = 26,27)
  ROUND A (f = 5060)
  ADD AL PLUS CONSTANT (f= 71)
  FLOATING POINT DIVIDE (f = 5005)

  The stage of the overflow designator is tested by either the SKIP ON OVERFLOW instruction (f = 5053). The execution of either instruction automatically clears the overflow designator. When the overflow designator is set, interrupts are not locked out.

- Guard Mode Designator

  The guard mode designator is a bi-stable, single-stage element set as a result of the LGM (f = 5065) instruction. It is cleared by the occurrence of any interrupt. While the guard mode designator is set, each instruction store cycle is checked. If the referenced address does not fall within the upper and lower storage limits, a guard mode interrupt is generated.

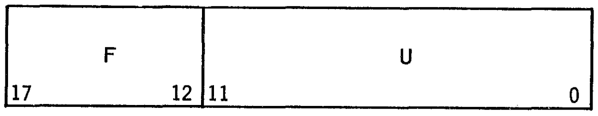
5.4. INSTRUCTION TYPES AND FORMATS

Instructions are binary numbers formatted in such a manner that when they are transferred to and interpreted by the command/arithmetic section of the computer, they result in the execution of a predefined operation. Instructions for the UNIVAC 418-III System are comprised of two entities, the function field and the operand field. The contents of the function field informs the c/a section which operation is to be performed; the contents of the operand field supplies the c/a section with the necessary information to enable it to perform the function. The set of all recognized functions is referred to as the instruction repertoire.

The UNIVAC 418-III instructions are divided into three distinct categories, referred to as type I, type II, and type III instructions. Type I instructions are identified by function codes 02 through 027, 032, 033, and 040 through 047. Type II instructions are identified by function codes 030, 031, 034 through 037, and 051 through 076. Type III instructions are identified by function codes 5000 through 5077.

- Type I instructions

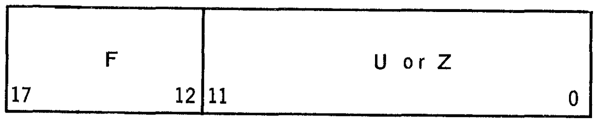
  The type I instruction format is:



where: F is the 6-bit function code.
U is the 12 low-order bits of the operand address.

- Type II instructions

  The type II instruction format is:



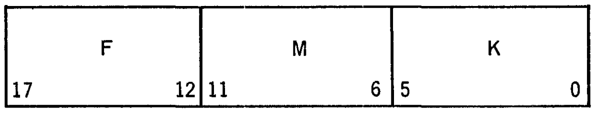
where: F is the 6-bit function code.
U is the 12 low-order bits of the operand address.
Z is the 12 low-order bits of an 18-bit sign extended operand.

When F indicates that the 12 low-order bits are to be interpreted as the actual operand, an 18-bit operand is formed by using Z and propagating the contents of bit 11 to the high-order 6 bits. This is commonly referred to as sign extension.

- Type III instructions

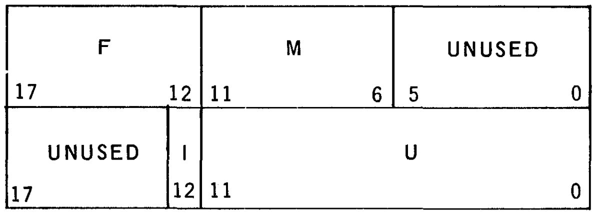
  Type III instructions may be divided into two distinct categories, each with a slightly different format. They are all categorized by a major function code of 050, and a minor function code between 0 and 077.

Type III-a



where: F is 50_8.
M is the minor function code.
K is 0 or a constant less than 64.

Type III-b



where: F is 50_8.
M is the minor function code.
I is 0 or 1 depending on whether indexing is to be used.
U is the 12 low-order bits of the operand address.

Note that the type III-b instructions are two-word (36-bit) instructions. In addition to the above formats there are several type III-a instructions which use the contents of one or more memory locations following their occurrence for specific data. These are principally the I/O instructions. They transfer control to the memory location following the data words used by them.

5.5. ADDRESSING

The operand fields of type I, type II, and type III-b instructions contain 12 bits. The UNIVAC 418-III main storage is logically divided into bays, each containing 4096 18-bit words, and may be expanded to a maximum of 32 bays; therefore, each type I, type II, or type III-b instruction provides sufficient space to specify any address within a bay. The bay which contains the desired address is determined by certain rules outlined in the following discussion.

When an instruction is executed which is in the last storage location of a bay, program control passes to the first location of the next bay unless it is a skip or jump type instruction. If it is a skip type instruction, control passes to the first or second location of the next bay depending on whether or not the skip condition is met. If it is a jump type instruction, control passes to the storage location specified in the next bay. This is tantamount to saying that as long as forward jumps are made, it does not matter where the instruction is located in storage.

In order to enable special-register-sensitive instructions to access any address in storage, the SR (special register) may be used to specify which bay is to be used. The special register is active or inactive depending on whether bit 4 is set to 1 or to 0; bit 4 is not a part of the bay identification. Bits 5 and 3 through 0 of SR are the bay bits.

Example:

To set the SR active to bay 25 (31_8), the binary number $\begin{array}{cccccc} 5&4&3&2&1&0 \\ \hline \multicolumn{1}{|c|}{1}&\multicolumn{1}{c|}{1}&\multicolumn{1}{c|}{1}&\multicolumn{1}{c|}{0}&\multicolumn{1}{c|}{0}&\multicolumn{1}{c|}{1} \\ \hline \end{array}$ (71_8) must be stored in SR because bit 4 (SR active bit) must be set to 1. The desired address is derived by ignoring bit 4 and treating bit 5 as though it were in bit position 4. By doing this, 71_8 becomes 31_8 ($111001_2 \rightarrow 11001_2$).

To set the SR active to bay 5 (5_8), the binary number $\begin{array}{cccccc} 5&4&3&2&1&0 \\ \hline \multicolumn{1}{|c|}{0}&\multicolumn{1}{c|}{1}&\multicolumn{1}{c|}{0}&\multicolumn{1}{c|}{1}&\multicolumn{1}{c|}{0}&\multicolumn{1}{c|}{1} \\ \hline \end{array}$ (25_8) must be stored in SR.

In order to set the special register active to bay 3, the instruction:

```
    LABEL           OPERATION                      OPERAND
1             10              20              30              40
              LSR             023

```

is executed. To set it active to bay 31 (32nd bay), the instruction:

```
              LSR             077


```

is executed. To inactivate the special register, the instruction:

```
              LSR             0


```

may be executed.

■ Type I Instructions

Type I instructions are SR-sensitive and indexable, meaning that if SR is active, the bay specified by its contents is accessed, and that the contents of the active index register are used to modify the operand address if the function code is odd.

If SR is not active (bit 4 is 0), the bay to be accessed is that in which the instruction itself resides; the bay bits are taken from the five high-order bits of the instruction address register.

If the function code (f) is odd, indexing is specified. This means that the full 18-bit contents of the active index register are arithmetically added to the (positive) 17-bit operand address. Figure 5–1 illustrates the various addressing techniques for type I instructions.

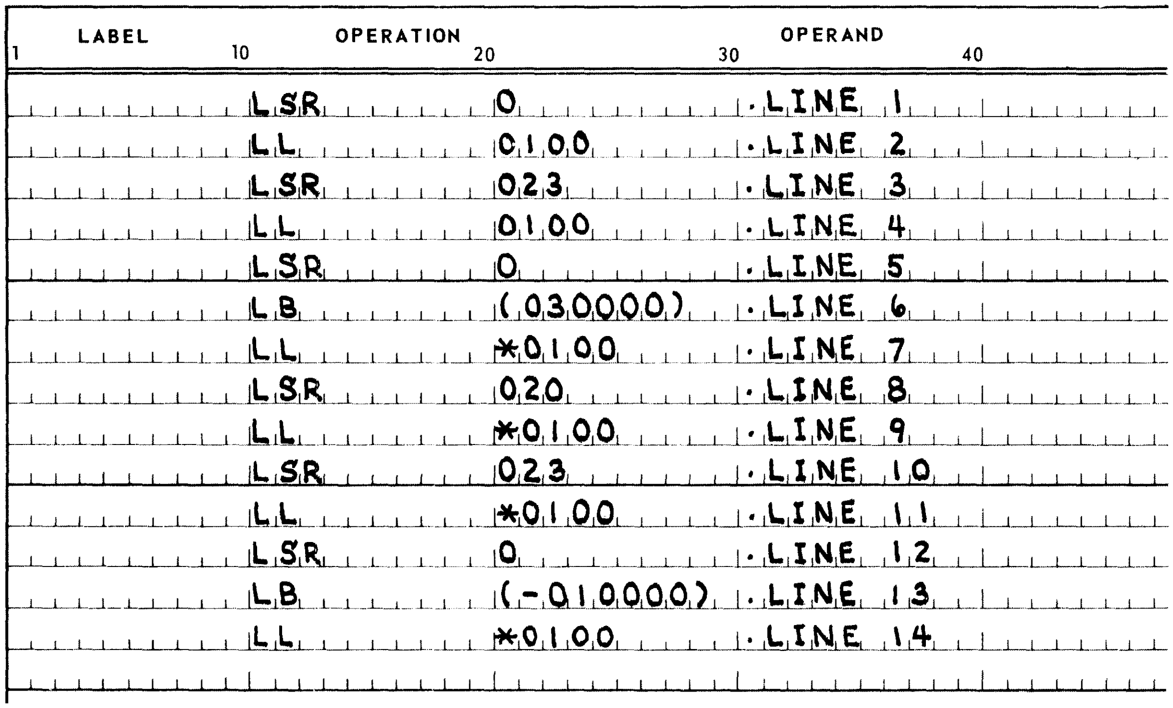

```
LABEL     OPERATION           OPERAND
1         10        20        30        40
          LSR       0         .LINE 1
          LL        0100      .LINE 2
          LSR       023       .LINE 3
          LL        0100      .LINE 4
          LSR       0         .LINE 5
          LB        (030000)  .LINE 6
          LL        *0100     .LINE 7
          LSR       020       .LINE 8
          LL        *0100     .LINE 9
          LSR       023       .LINE 10
          LL        *0100     .LINE 11
          LSR       0         .LINE 12
          LB        (-010000) .LINE 13
          LL        *0100     .LINE 14
```

Figure 5–1. Type I Instruction Addressing Techniques

If line 1 were to be located at address 020000, the following storage references would be made:

LINE NUMBER	EFFECTIVE U	ADDRESS REFERENCED
2	0100 + 020000	020100
4	0100 + 030000	030100
7	0100 + 020000 + 030000	050100
9	0100 + 000000 + 030000	030100
11	0100 + 030000 + 030000	060100
14	0100 + 020000 – 010000	010100

- Type II Instructions

Type II instructions are never SR-sensitive, differing in this respect from type I instructions. Regardless of the contents of SR, the bay referenced is the one in which the instruction resides.

Some type II instructions are index-sensitive; this allows them to access other bays by using the active index register to modify the address obtained by combining $U_{11\text{-}0}$ and $IAR_{17\text{-}12}$.

Three instructions (LBK, LLK, and ALK) do not make a second storage access. The sign-extended value of the operand field is used as the operand.

- Type III Instructions

  The type III-a instructions do not require an operand. The type III-b instructions resemble the type I instructions; they are SR- and index-sensitive. When I is set to 1, indexing is used; when it is set to 0, no indexing is used.

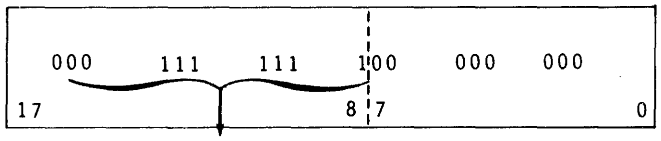
5.6. STORAGE PROTECTION (GUARD MODE LIMITS)

To ensure program protection, a selected area of storage may be placed under guard mode limits through the use of the LGM (f = 5065) instruction. When the guard mode is active, any attempt to store into a storage address outside the range set by the LGM instruction causes a guard mode interrupt at address 30_8. Two nine-bit registers, storage limits upper and storage limits lower, may be loaded with the upper and lower bounds of an area of storage to be placed under guard mode. For this purpose, storage is divided into 256-word blocks. The LGM is a privileged instruction and may not be used by the programmer.

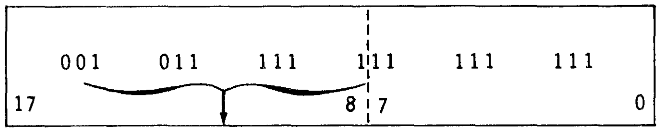
When the nine high-order bits of a 17-bit storage address are placed in storage limits lower, the first address of that block is the lower bound of the guard mode limits. When the nine high-order bits of a 17-bit storage address are placed in storage limits upper, the last address of that block is the upper bound of the guard mode limits. For example, the instruction:

```
 LABEL            OPERATION                OPERAND
1          10                  20            30             40
          LGM
          +0277177
```

prevents storage outside the range of addresses 077400_8 to 0137777_8; any attempted violation of this restriction causes a guard mode interrupt instead.



0177 = Storage-Limits-Lower Contents



0277 = Storage-Limits-Upper Contents

Upon the occurrence of any interrupt, the guard mode designator is cleared (disabled), so that all of main storage becomes accessible to subroutines gaining control through the interrupt locations.

Because locations 0 through 17_8 are never under guard mode protection, it is always possible to use them for storage. The index registers are part of that category and are actually located at addresses 1 through 10_8.

5.7. PRIVILEGED INSTRUCTIONS

Privileged instructions are those which are needed by an operating (controlling) system in order to perform its job; they are considered inappropriate for use in normal (user) programs. The appearance of any of these instructions in any user program would have an unpredictable and probably disastrous effect.

When the guard mode designator is set, through the use of an LGM instruction, any attempt to execute a privileged instruction causes a guard mode interrupt instead. The privileged instruction is not executed or initiated.

The privileged instructions are:

5011	load input channel (LIC)
5012	load output channel (LOC)
5013	load external function channel (LFC)
5015	stop input on channel (STIC)
5016	stop output on channel (STOC)
5021	test input channel (TIC)
5022	test output channel (TOC)
5023	test function channel (TFC)
5024	wait for interrupt (WFI)
5025	wait for interrupt (WFI)
5056	stop on key setting (SK) (ignored when in guard mode)
5065	load guard mode (LGM)
5066	set audible alarm (SSA)
5067	enable ESI interrupts (EEI)

5.8. FLOATING-POINT NUMBERS

Floating-point numbers are two-word, 36-bit constants; they consist of a fixed-point part (mantissa) and an exponent (characteristic). The format of a floating point number is:

s	c	m	
35	34 27	26 18	17 0

where:

s is the sign bit.
c is the eight characteristic bits.
m is the 27 mantissa bits.

The mantissa (m) contains the 27 significant bits of the floating-point number. The magnitude of the mantissa is either 0 or between $.4_8$ and $.777777777_8$, normalized so that the most significant bit is a 1. The characteristic is the value of c in the expression $2^{c-200_8}*m$. The high-order bit of c (bit 34) is the sign bit of the characteristic. When $c_{34}=1$, the characteristic is positive; when $c_{34}=0$, the characteristic is negative. The sign bit (s) is 0 when the floating-point number is greater than 0 (positive); it is 1 when the number is less than 0 (negative). The magnitude (positive equivalent) of a negative number is its one's complement.

For example, the number 2.0 can be rewritten in floating-point form as:

$$2.0 * 10^0, \text{ or}$$
$$20. * 10^{-1}, \text{ or}$$
$$.20 * 10^1, \text{ and many others.}$$

In these examples, 0, −1, and 1 are the characteristics; 2.0, 20., and .20 are the mantissas. The three expressions represent the same quantities, illustrating that the mantissa and characteristic may be manipulated so that the value of the number remains unchanged. The octal representation of this number is:

$$2.0 * 10^0 = .2_8 * 2^3$$

To normalize, the mantissa is multiplied by 2, and the characteristic is decreased by 1.

The floating-point format is.

$$002400000000_8$$

Finally, to indicate that the power of the characteristic is positive, the characteristic is biased to obtain 202400000000_8. In the same manner, −2.0 is represented as 575377777777_8.

5.9. INTERRUPTS

Interrupts are internally generated signals which cause the c/a section to interrupt its normal sequence of instructions (governed by instruction address register contents), and to take the next instruction from a predetermined address in main storage. The contents of the IAR are not changed until the interrupt instruction is executed. An SLJ or SLJI instruction is placed in the interrupt locations, which captures the value of IAR in order to allow normal processing to continue when the interrupt processing coding is completed.

6. INSTRUCTION REPERTOIRE DESCRIPTION

6.1. SYMBOL CONVENTIONS

The following is a list of the ''shorthand'' symbols used in the repertoire description. The meaning each symbol conveys appears to the right of the symbol.

AU	Upper accumulator, 18-bit arithmetic register
AL	Lower accumulator, 18-bit arithmetic register
A	AU and AL linked together to form one 36-bit arithmetic register
B	Eight index registers with seven residing in main storage and the currently active index register in a flip-flop register
f	Function code, six high-order bits of all instruction words
F	Function register; seven bits
k	Designator contained in type III instruction words; six bits
m	Minor function code contained in type III instruction words; six bits
M	(y), [y + (B)], [(y) **AND** (AU)], or [(y + (B)) **AND** (AU)] of compare instructions
NI	Next instruction
P or IAR	Program address register; 17 bits (or instruction address register)
SR	Special register; five bits, plus one active bit
IRP	Index register pointer; 3 bits
U	12 low-order bits contained in type I and type II instructions
U_P	U prefaced with the core storage segment designator bits of P (P_{16-12})
U_{SR}	U prefaced with the core storage segment designator bits of SR ($SR_{5,3-0}$)
y	Either an address formed by U_P or U_{SR} plus U_{11-0} or a constant formed by U with sign extension.
( )	Contents of an address or register
$( )_i$	Initial contents of an address or register
$( )_f$	Final contents of an address or register
$(_n)$	Contents of the nth bit of a register
(y–1,y)	Contents of two consecutive memory locations linked together to form a 36-bit word. Address y – 1 contains the most significant half of the word; y contains the least significant half of the word.
:	Indicates COMPARISON when used in logical expressions.
( ) **AND** ( )	Bit-by-bit or logical product (logical AND) defined by the following table:

	0	1
0	0	0
1	0	1

( ) **OR** ( ) Logical sum (inclusive OR) defined by the following table:

	0	1
0	0	1
1	1	1

( )'	One's complement of the contents of an address or register
( ) · ( )	Algebraic product of the contents of two locations
→	Transfer of the quantity stated at the left of the symbol to the address or register stated at the right of the symbol
[ ]	Used to group terms. The brackets do not indicate "the contents of".

6.2. INSTRUCTION REPERTOIRE

The instruction repertoire for the UNIVAC 418-III Assembler is described in this section. The instructions are listed and defined in the following format:

Octal Code	Instruction Name	Mnemonic

Operation performed (Symbolic summary)
Definition of the y address or constant
Test defining the instruction
Examples or notes, where necessary

Common usage and example cases are included where necessary to supplement the description; however, no attempt is made in these descriptions to indicate more sophisticated uses for any of the instructions.

6.2.1. Supervisor Call Instructions

Several function codes are not assigned a specific function. These are called supervisor call instructions because when executed they cause a supervisor call interrupt at location 20_8. Depending on software conventions, the RTOS may perform certain software functions when encountering these illegal function codes.

The supervisor call functions are:

00,01,077

Execution Time: 0.75 usec, and

5000,5001,5077

Execution Time: 1.00 usec.

6.3. TYPES I AND II INSTRUCTIONS

02 COMPARE LOWER (CL)

Operation: (AL): (y)

Execution Time: 1.50 usec.

$y = U_P$ or $U_{SR} + U_{11-0}$

The COMPARE stage of the compare designator is set.

This instruction compares the contents of AL algebraically with the contents of y and the compare designator is set as follows:

1. The LESS THAN stage is set if $(AL) < (y)$.
2. The EQUAL stage is set if $(AL) = (y)$.

The contents of AL remain unchanged and in AL. $(AL)_f = (AL)_i$.

NOTES:

- $-0 < +0$
- The COMPARE stage is cleared by the execution of any instruction other than the arithmetic jump instructions ($f = 6067$). Thus, if the result of a COMPARE instruction is to be successfully tested, it must be immediately followed by one or more of the conditional jump instructions.
- Arithmetic jump instructions have significantly different operations if executed when the COMPARE stage is not set.
- When the COMPARE stage of the compare designator is set, all interrupts are locked out to avoid the possibility of inadvertently clearing the COMPARE stage.

03 COMPARE LOWER (CL*)

Operation: (AL) : (y + (B))

Execution Time: 1.50 usec.

$y = U_P$ or $U_{SR} + U_{11-0}$

The COMPARE stage of the compare designator is set.

This instruction compares the contents of AL algebraically with the contents of y + (B) and the compare designator is set as follows:

1. The LESS THAN stage is set if $(AL) < (y + (B))$.
2. The EQUAL stage is set if $(AL) = (y + (B))$.

The contents of AL remains unchanged and in AL. $(AL)_f = (AL)_i$.

NOTES:

- $-0 < +0$
- The COMPARE stage is cleared by the execution of any instruction other than the arithmetic jump instructions ($f = 6067$). Thus, if the result of a COMPARE instruction is to be successfully tested, it must be immediately followed by one or more of the conditional jump instructions.
- Arithmetic jump instructions have significantly different operation if executed when the COMPARE stage is not set.
- When the COMPARE stage of the compare designator is set, all interrupts are locked out to avoid the possibility of inadvertently clearing the COMPARE stage.

04 MASKED SELECTIVE LOAD (MSL)

Operation: [(AU) AND (AL)] OR [(AU) AND (y)] $\rightarrow$ AL

Execution Time: 1.50 usec.

$y = U_P$ or $U_{SR} + U_{11\text{-}0}$

This instruction replaces the individual bits of AL with bits of the contents of y corresponding to 1's in AU, leaving the remaining bits of AL unaltered. If $(AU)_n = 1$, then $(y)_n \rightarrow AL_n$.

The contents of AU remain unchanged and in AU. $(AU)_f = (AU)_i$.

Example: $(AU)_i$ = 007777 – Mask
(y) = 123451
$(AL)_i$ = 666666
$(AL)_f$ = 663451

NOTES:

- A mask of positive zero does not change AL. $(AL)_f = (AL)_i$
- A mask of negative zero results in the transfer of the contents of y to AL. $(AL)_f = (y)$

05 MASKED SELECTIVE LOAD (MSL*)

Operation: [(AU) AND (AL)] OR [(AU) AND (y + (B))] $\rightarrow$ AL

Execution Time: 1.50 usec.

$y = U_P$ or $U_{SR} + U_{11\text{-}0}$

This instruction replaces the individual bits of AL with bits of the contents of y + (B) corresponding to 1's in AU, leaving the remaining bits of AL unaltered. If $(AU)_n = 1$, then $(y + (B))_n \rightarrow AL_n$.

The contents of AU remain unchanged and in AU. $(AU)_f = (AU)_i$

NOTES:

- A mask of positive zero does not change AL. $(AL)_f = (AL)_i$
- A mask of negative zero results in the transfer of the contents of y + (B) to AL. $(AL)_f = (y + (B))$

06 COMPARE LOWER MASKED BY UPPER (CLM)

Operation: [(AU) AND (AL)] : [(AU) AND (y)]

Execution Time: 2.00 usec.

$y = U_P$ or $U_{SR} + U_{11\text{-}0}$

The COMPARE stage of the compare designator is set.

This instruction compares selected bits of AL with corresponding bits of the contents of y by logically multiplying AU by AL and by the contents of y and algebraically comparing the two resultants. The compare designator is set as follows:

1. The LESS THAN stage is set if [(AL) AND (AU)] < [(y) AND (AU)]

2. The EQUAL stage is set if [(AL) AND (AU)] = [(y) AND (AU)]

The contents of AL remain unchanged and in AL. The contents of AU remain unchanged and in AU. $(AL)_f = (AL)_i$ and $(AU)_f = (AU)_i$

Example:

$(AU)_i$ = 007777 – Mask
(y) = 123451
$(AL)_i$ = 222351
COMPARE 2351 with 3451
$(AU)_f$ = 007777
$(AL)_f$ = 222351

NOTES:

- $-0 < +0$

- The COMPARE stage is cleared by the execution of any instruction other than the arithmetic jump instructions (f = 6067). Thus, if the result of a COMPARE instruction is to be successfully tested, it must be immediately followed by one or more of the conditional jump instructions.

- Arithmetic jump instructions have significantly different operations if executed when the COMPARE stage is not set.

- When the COMPARE stage of the compare designator is set, all interrupts are locked out to avoid the possibility of inadvertently clearing the COMPARE stage.

07 COMPARE LOWER MASKED BY UPPER (CLM*)

Operation: [(AU) AND (AL)] : [(AU) AND (y + (B))]

Execution Time: 2.00 usec.

$y = U_P$ or $U_{SR} + U_{11\text{-}0}$

The COMPARE stage of the compare designator is set.

This instruction compares selected bits of AL with corresponding bits of the contents of y + (B) by logically multiplying AU by AL and by the contents of y + (B) and algebraically comparing the two resultants. The compare designator is set as follows:

1. The LESS THAN stage is set if [(AL) AND (AU)] < [(y + (B)) AND (AU)]
2. The EQUAL stage is set if [(AL) AND (AU)] = [(y + (B)) AND (AU)]

The contents of AL remain unchanged and in AL. The contents of AU remain unchanged and in AU. $(AL)_f = (AL)_i$ and $(AU)_f = (AU)_i$.

NOTES:

- $-0 < +0$
- The COMPARE stage is cleared by the execution of any instruction other than the arithmetic jump instructions (f = 6067). Thus, if the result of a COMPARE instruction is to be successfully tested, it must be immediately followed by one or more of the conditional jump instructions.
- Arithmetic jump instructions have significantly different operations if executed when the COMPARE stage is not set.
- When the COMPARE stage of the compare designator is set, all interrupts are locked out to avoid the possibility of inadvertently clearing the COMPARE stage.

10 LOAD AU (LU)

Operation: $(y) \rightarrow AU$

Execution Time: 1.50 usec.

$y = U_P$ or $U_{SR} + U_{11\text{-}0}$

Clear AU.

This instruction transfers the contents of y to AU.

The contents of y remain unchanged and in y. $(y)_f = (y)_i$

11 LOAD AU (LU*)

Operation: $(y + (B)) \rightarrow AU$

Execution Time: 1.50 usec.

$y = U_P$ or $U_{SR} + U_{11\text{-}0}$

Clear AU.

This instruction transfers the contents of y + (B) to AU.

The contents of y + (B) remain unchanged and in y + (B). $(y + (B))_f = (y + (B))_i$

12 LOAD AL (LL)

Operation: $(y) \rightarrow AL$

Execution Time: 1.50 usec.

$y = U_P$ or $U_{SR} + U_{11\text{-}0}$

Clear AL.

This instruction transfers the contents of y to AL.

The contents of y remain unchanged and in y. $(y)_f = (y)_i$

13 LOAD AL (LL*)

Operation: $(y + (B)) \rightarrow AL$

Execution Time: 1.50 usec.

$y = U_P$ or $U_{SR} + U_{11\text{-}0}$

Clear AL.

This instruction transfers the contents of y + (B) to AL.

The contents of y + (B) remain unchanged and in y + (B). $(y + (B))_f = (y + (B))_i$

14 ADD TO LOWER (AL)

Operation: $[(y) + (AL)] \rightarrow AL$

Execution Time: 1.50 usec.

$y = U_P$ or $U_{SR} + U_{11\text{-}0}$

This instruction adds the contents of y to the contents of AL and places the resultant, SUM, in AL.

The contents of y remain unchanged and in y. $(y)_f = (y)_i$

NOTES:

- If the contents of AL is negative 0 and the contents of y is negative 0, the result of the addition is negative 0.

  $(AL)_f$ = 1's if $(AL)_i$ = 1's and (y) = 1's

- The results of addition involving all other possible combinations of positive and negative 0 are positive 0.
- If the magnitude of the resultant is too large for AL to hold, that is, the sum exceeds the range -377777_8 to $+377777_8$, the result is incorrect and the overflow designator is set. The state of the overflow designator is tested by either the SKIP ON OVERFLOW instruction (f = 5052) or the SKIP ON NO OVERFLOW instruction (f = 5053). The execution of either of these two instructions clears the overflow designator.

15 ADD TO LOWER (AL*)

Operation: $[(y + (B)) + (AL)] \rightarrow AL$

Execution Time: 1.50 usec.

$y = U_P$ or $U_{SR} + U_{11-0}$

This instruction adds the contents of y + (B) to the contents of AL and stores the SUM in AL.

The contents of y + (B) remain unchanged and in y + (B). $(y + (B))_f = (y + (B))_i$

NOTES:

- If the contents of AL is negative 0 and the contents of y + (B) is negative 0, the result of the addition is negative 0.
- The results of addition involving all other possible combinations of positive and negative 0 are positive 0.
- If the magnitude of the resultant is too large for AL to hold, that is, the sum exceeds the range -377777_8 to $+377777_8$, the result is incorrect and the overflow designator is set. The state of the overflow designator is tested by either the SKIP ON OVERFLOW instruction (f = 5052) or the SKIP ON NO OVERFLOW instruction (f = 5053). The execution of either of these two instructions clears the overflow designator.

16 ADD NEGATIVELY TO LOWER (ANL)

Operation: $[(AL) - (y)] \rightarrow AL$

Execution Time: 1.50 usec.

$y = U_P$ or $U_{SR} + U_{11-0}$

This instruction subtracts the contents of y from the contents of AL and places the resultant, DIFFERENCE, in AL.

The contents of y remain unchanged and in y. $(y)_f = (y)_i$

NOTES:

- If the contents of AL is negative 0 and the contents of y is positive 0, the result of the subtraction is negative 0. $(AL)_f$ = 1's if $(AL)_i$ = 1's and (y) = 0's.

- The results of subtraction involving all other possible combinations of positive and negative 0 are positive 0.

- If the magnitude of the resultant is too large for AL to hold, that is, the difference exceeds the range -377777_8 to $+377777_8$, the result is incorrect and the overflow designator is set. The state of the overflow designator is tested by either the SKIP ON OVERFLOW instruction (f = 5052) or the SKIP ON NO OVERFLOW instruction (f = 5053). The execution of either of these two instructions clears the overflow designator.

17 ADD NEGATIVELY TO LOWER (ANL*)

Operation: $[(AL) - (y + (B))] \rightarrow AL$

Execution Time: 1.50 usec.

$y = U_P$ or $U_{SR} + U_{11-0}$

This instruction subtracts the contents of y + (B) from the contents of AL and places the resultant, DIFFERENCE, in AL.

The contents of y + (B) remain unchanged and in y + (B). $(y + (B))_f = (y + (B))_i$

NOTES:

- If the contents of AL is negative 0, and the contents of y + (B) is positive 0, the result of the subtraction is negative 0. $(AL)_f$ = 1's if $(AL)_i$ = 1's and (y + (B)) = 0's.

- The results of subtraction involving all other possible combinations of positive and negative 0 are positive 0.

- If the magnitude of the resultant is too large for AL to hold, that is, the difference exceeds the range -377777_8 to $+377777_8$, the result is incorrect and the overflow designator is set. The state of the overflow designator is tested by either the SKIP ON OVERFLOW instruction (f = 5052) or SKIP ON NO OVERFLOW instruction (f = 5053). The execution of either of these two instructions clears the overflow designator.

20 ADD TO A (AA)

Operation: $[(A) + (y-1,y)] \rightarrow A$

Execution Time: 3.00 usec.

$y = U_P$ or $U_{SR} + U_{11-0}$

The borrow designator is cleared to zero.

This instruction is executed by combining the AU and AL registers into a 36-bit accumulator, the A register. The contents of y–1 and y are treated as one 36-bit word, a double-length signed binary number. The contents of y–1, y are added to the contents of A and the resultant, SUM, is placed in A.

The contents of y–1, y remain unchanged and in y–1, y. $(y-1,y)_f = (y-1,y)_i$

Example:

```
y = 07507
(A)i    = 201007430145
(07507) =       351123 (least significant half)
(07506) = 077430       (most significant half)
          ____________
(A)f    = 300440001270
```

NOTES:

- The least significant half of the 36-bit number is in y; the most significant half of the 36-bit number is in y–1. The sign of the 36-bit double-length number is indicated by the most significant bit of (y–1).
- The operating characteristics of double-length arithmetic operations are the same as those for single-length arithmetic operations, except that any borrow for AL comes from AU.
- If an end-around borrow for AU is required, it is inhibited and the borrow designator is set, indicating that the result left in A is too large by 1 and must be corrected. This condition is tested by the TEST NO BORROW instruction (f = 5051). The borrow designator is cleared only by the execution of another ADD TO A (f = 20,21) or ADD NEGATIVE TO A (f = 22,23) instruction.
- If the contents of A is negative 0 and the contents of y–1,y is negative 0, the result of the addition is negative 0. $(A)_f$ = 1's if $(A)_i$ = 1's and (y–1,y) = 1's
- The results of addition involving all other possible combinations of positive and negative 0 are positive 0.
- If the magnitude of the resultant is too large for A to hold, that is, the sum exceeds the range -377777777777_8 to $+377777777777_8$, the result is incorrect and the overflow designator is set. The state of the overflow designator is tested by either the SKIP NO OVERFLOW instruction (f = 5052) or the SKIP ON NO OVERFLOW instruction (f = 5053). The execution of either of these two instructions clears the overflow designator.

21 ADD TO A (AA*)

Operation: $[(A) + (y + (B) -1, y + (B))] \rightarrow A$

Execution Time: 3.00 usec.

$y = U_P$ or $U_{SR} + U_{11\text{-}0}$

The borrow designator is cleared to zero.

This instruction is executed by combining the AU and AL registers into a 36-bit accumulator, the A register. The contents of y + (B) and y + (B)–1 are treated as one 36-bit word, a double-length signed binary number. The contents of y + (B)–1, y + (B) are added to the contents of A and the resultant, SUM, is placed in A.

The contents of y + (B)–1, y + (B) remain unchanged and in y + (B)–1, y + (B). $(y + (B)-1, y + (B))_f = (y + (B)-1, y + (B))_i$

NOTES:

- The least significant half of the 36-bit number is in y + (B); the most significant half of the 36-bit number is y + (B)–1. The sign of the 36-bit double-length number is indicated by the most significant bit of (y + (B)–1).

- The operating characteristics of double-length arithmetic operations are the same as those for single-length arithmetic operations, except that any borrow for AL comes from AU.

- If an end-around borrow for AU is required, it is inhibited, and the borrow designator is set indicating that the result left in A is too large by 1 and must be corrected. This condition is tested by the TEST NO BORROW instruction (f = 5051). The borrow designator is cleared only by the execution of another ADD TO A (f = 20,21) or ADD NEGATIVELY TO A (f = 22,23) instruction.

- If the contents of A is negative 0 and the contents of y–1,y is negative 0, the result of the addition is negative 0. $(A)_f$ = 1's if $(A)_i$ = 1's and (y + (B)–1, y + (B)) = 1's

- The results of addition involving all other possible combinations of positive and negative 0 are positive 0.

- If the magnitude of the resultant is too large for A to hold, that is, the sum exceeds the range -377777_8 to $+377777_8$, the result is incorrect and the overflow designator is set. The state of the overflow designator is tested by either the SKIP ON OVERFLOW instruction (f = 5052) or the SKIP ON NO OVERFLOW instruction (f = 5053). The execution of either of these two instructions clears the overflow designator.

22 ADD NEGATIVELY TO A (ANA)

Operation: $[(A) - (y-1,y)] \rightarrow A$

Execution Time: 3.00 usec.

$y = U_P$ or $U_{SR} + U_{11-0}$

The borrow designator is cleared.to zero.

This instruction is executed by combining the AU and AL registers into a 36-bit accumulator, the A register. The contents of y–1 and y are treated as one 36-bit word, a double-length signed binary number. The contents of y–1,y are subtracted from the contents of A and the resultant, DIFFERENCE, is placed in A.

The contents of y–1,y remain unchanged and in y–1,y. $(y-1,y)_f = (y-1,y)_i$

Example:

y = 07507		
$(A)_i$ =	201007430145	
(07507) =	351123	(least significant half)
(07506) =	077430	(most significant half)
$(A)_f$ =	101357057022	

NOTES:

- The least significant half of the 36-bit number is in y; the most significant half of the 36-bit number is in y–1. The sign of the 36-bit double-length number is indicated by the most significant bit of (y–1).
- The operating characteristics of double-length arithmetic operations are the same as those for single-length arithmetic operations, except that any borrow for AL comes from AU.
- If an end-around borrow for AU is required, it is inhibited and the borrow designator is set, indicating that the result left in A is too large by 1 and must be corrected. This condition is tested by the TEST NO BORROW instruction (f = 5051). The borrow designator is cleared only by the execution of another ADD TO A (f = 20,21) or ADD NEGATIVELY TO A (f = 22,23) instruction.
- If the contents of A is negative 0 and the contents of y–1,y is positive 0, the result of the subtraction is negative 0. $(A)_f$ = 1's if $(A)_i$ = 1's and (y–1,y) = 0's
- The results of subtraction involving all other possible combinations of positive and negative 0 are positive 0.

- If the magnitude of the resultant is too large for A to hold, that is, the difference exceeds the range -377777777777_8 to $+377777777777_8$, the result is incorrect and the overflow designator is set. The state of the overflow designator is tested by either the SKIP ON OVERFLOW instruction (f = 5052) or the SKIP ON NO OVERFLOW instruction (f = 5053). The execution of either of these two instructions clears the overflow designator.

23 ADD NEGATIVELY TO A (ANA*)

Operation: $[(A) - (y + (B) - 1,\ y + (B)] \rightarrow A$

Execution Time: 3.00 usec.

$y = U_P$ or $U_{SR} + U_{11\text{-}0}$

The borrow designator is cleared to zero.

This instruction is executed by combining the AU and AL registers into a 36-bit accumulator, the A register. The contents of y + (B) – 1 and y + (B) are treated as one 36-bit word, a double-length signed binary number. The contents of y + (B)–1, y + (B) are subtracted from the contents of A and the resultant, DIFFERENCE, is placed in A.

The contents of y + (B)–1, y = (B) remain unchanged and in y + (B)–1, y + (B). $(y + (B)-1,\ y + (B))_f = (y + (B)-1,\ y + (B))_i$

NOTES:

- The least significant half of the 36-bit number is in y + (B); the most significant half of the 36-bit number is in y + (B)–1. The sign of the 36-bit double-length number is indicated by the most significant bit of (y + (B)–1).
- The operating characteristics of double-length arithmetic operations are the same as those for single-length arithmetic operations, except that any borrow for AL comes from AU.
- If an end-around borrow for AU is required, it is inhibited, and the borrow designator is set indicating that the result left in A is too large by 1 and must be corrected. This condition is tested by the TEST NO BORROW instruction (f = 5051). The borrow designator is cleared only by the execution of another ADD TO A or ADD NEGATIVELY TO A instruction.
- If the contents of A is negative 0 and the contents of y + (B)–1, y + (B) is positive 0, the result of the subtraction is negative 0. $(A)_f$ = 1's if $(A)_i$ = 1's and (y–1,y) = 0's.
- The results of subtraction involving all other possible combinations of positive and negative 0 are positive 0.
- If the magnitude of the resultant is too large for A to hold, that is, the difference exceeds the range -377777777777_8 to $+377777777777_8$, the result is incorrect and the overflow designator is tested by either the SKIP ON OVERFLOW instruction (f = 5052) or the SKIP ON NO OVERFLOW instruction (f = 5053). The execution of either of these two instructions clears the overflow designator.

24 MULTIPLY (M)

Operation: $[(AL) \times (y)] \rightarrow A$

Execution Time: 6.50 usec. – Numbers of like signs
7.375 usec. – Numbers of unlike signs

$y = U_P$ or $U_{SR} + U_{11\text{-}0}$

This instruction multiplies the contents of AL by the contents of y and the resultant, PRODUCT, is placed in the 36-bit accumulator, the A register, consisting of AU and AL.

The contents of y remain unchanged and in y. $(y)_f = (y)_i$

NOTES:

- The results of multiplication involving all possible combinations of positive and negative 0 are positive 0.

- If the most significant half of the product is 17 bits or smaller, it is contained in AL with leading 0's in cases of positive products and leading 1's in cases of negative products. AL_{17} contains the proper sign.

  Examples:

  Positive Product
  (AL) $000003_8 = +3$
  (y) $000004_8 = +4$
  (A) = (AU) + (AL) = $000000_8 + 000014_8$

  Negative Product
  (AL) = $777774_8 = -3$
  (y) = $000004_8 = +4$
  (A) = (AU) + (AL) = $777777_8 + 777763_8$

- If the most significant half of the product is exactly 18 bits long, it fills AL and the sign is carried by AU. For positive products, AU contains all 0's; for negative products, AU contains all 1's. AL_{17} does not contain the proper sign but, rather, the most significant bit of the product.

  Examples:

  Positive Product
  (AL) = 000725_8
  (y) = 000741_8
  (A) = (AU) + (AL) = $000000_8 + 670465_8$

  Negative Product
  (AL) = $777052_8 = -725_8$
  (y) = 000741_8
  (A) = (AU) + (AL) = $777777_8 + 107312_8$

- No overflow is possible with this instruction because the number of bits in the product cannot exceed the number of bits in the multiplicand plus the number of bits in the multiplier.

25 MULTIPLY (M*)

Operation: $[(AL) \times (y + (B))] \rightarrow A$

Execution Time: 6.50 usec. – Numbers of like signs
7.375 usec. – Numbers of unlike signs

$y = U_P$ or $U_{SR} + U_{11\text{-}0}$

This instruction multiplies the contents of AL by the contents of y + (B) and the resultant, PRODUCT, is placed in the 36-bit accumulator, the A register, consisting of AU and AL.

The contents of y + (B) remain unchanged and in y + (B). $(y + (B))_f = (y + (B))_i$

NOTES:

- The results of multiplication involving all possible combinations of positive and negative 0 are positive 0.
- If the most significant half of the product is 17 bits or smaller, it is contained in AL with leading 0's in cases of positive products and leading 1's in cases of negative products. AL_{17} contains the proper sign.
- If the most significant half of the product is exactly 18 bits long, it fills AL and the sign is carried by AU. For positive products, AU contains all 0's; for negative products, AU contains all 1's. AL_{17} does not contain the proper sign but, rather, the most significant bit of the product.
- No overflow is possible with this instruction because the number of bits in the product cannot exceed the number of bits in the multiplicand plus the number of bits in the multiplier.

26 DIVIDE (D)

Operation: $[(A) \div (y)] \rightarrow AL$; Remainder $\rightarrow AU$

Execution Time: 6.50 usec. – Numbers of like signs
7.375 usec. – Numbers of unlike signs

$y = U_P$ or $U_{SR} + U_{11\text{-}0}$

This instruction divides the contents of A by the contents of y. The QUOTIENT is placed in AL and the REMAINDER is placed in AU.

The contents of y remain unchanged and in y. $(y)_f = (y)_i$

NOTES:

- The results of division involving all possible combinations of positive and negative 0 are positive 0.
- The remainder always bears the sign of the dividend with the results satisfying the relationship: DIVIDEND = QUOTIENT × DIVISOR + REMAINDER
- If the dividend and the divisor have like signs, the quotient is positive. If they have unlike signs, the quotient is negative.

  Examples:

Divisor	Dividend	Quotient	Remainder
+4	+5	+1	+1
−4	+5	− 1	+1
+4	−5	−1	−1
−4	−5	+1	−1

- If the magnitude of the quotient is too large for AL to hold, that is, the quotient exceeds the range -377777_8 to $+377777_8$, the result is incorrect and the overflow designator is set. The state of the overflow designator is tested by either the SKIP ON OVERFLOW instruction (f = 5052) or the SKIP ON NO OVERFLOW instruction (f = 5053). The execution of either of these two instructions clears the overflow designator.

27 DIVIDE (D*)

Operation: $[(A) \div (y + (B))] \rightarrow$ AL; Remainder $\rightarrow$ AU

Execution Time: 6.50 usec. – Numbers of like signs
7.375 usec. – Numbers of unlike signs

$y = U_P$ or $U_{SR} + U_{11\text{-}0}$

This instruction divides the contents of A by the contents of y + (B). The QUOTIENT is placed in AL and the REMAINDER is placed in AU.

The contents of y remain unchanged and in y. $(y + (B))_f = (y + (B))_i$

NOTES:

- The results of division involving all possible combinations of positive and negative 0 are positive 0.
- The remainder always bears the sign of the dividend with the results satisfying the relationship: DIVIDEND = QUOTIENT × DIVISOR + REMAINDER
- If the dividend and the divisor have like signs, the quotient is positive. If they have unlike signs, the quotient is negative.

Examples:

Divisor	Dividend	Quotient	Remainder
+4	+5	+1	+1
–4	+5	–1	+1
+4	–5	–1	–1
–4	–5	+1	–1

- If the magnitude of the quotient is too large for AL to hold, that is, the quotient exceeds the range -377777_8 to $+377777_8$, the result is incorrect and the overflow designator is set. The state of the overflow designator is tested by either the SKIP ON OVERFLOW instruction (f = 5052) or SKIP ON NO OVERFLOW instruction (f = 5053). The execution of either of these two instructions clears the overflow designator.

30 STORE LOCATION AND JUMP INDIRECT (SLJI)

Operation: $[(P) + 1] \rightarrow (y); [(y) + 1] \rightarrow P$

Execution Time: 2.25 usec.

$y = U_P + U_{11-0}$

This instruction stores the current program address +1 at the address defined by the contents of y. The contents of y are increased by 1, and the new address is transferred to the P register.

Example of an indirect return jump executed from address 002000_8:

ADDRESS	INITIAL CONTENTS	FINAL CONTENTS	EXPLANATION
002000_8	30 6500_8	30 6500_8	Execute subroutine from main program
006500_8	71 7420_8	71 7420_8	Constant defining location of desired subroutine
317420_8	37 2164_8	00 2001_8	Subroutine exit address
317421_8	———	00 2001_8	Subroutine entrance address (control is transferred here from indirect return jump)

The effect of the above sequence upon execution of the indirect return jump at address 002000_8 is to transfer control to the subroutine starting at 17421_8, while at the same time letting the subroutine know where to return control.

NOTE:

This instruction together with the jump indirect instruction provides the means needed for jumping to and from subroutines.

31 STORE LOCATION AND JUMP INDIRECT (SLJI*)

Operation: $[(P) + 1] \rightarrow (y + (B)); [(y + (B)) + 1] \rightarrow P$

Execution Time: 2.25 usec.

$y = U_P + U_{11-0}$

This instruction stores the current program address +1 at the address defined by the contents of y + (B). Then the contents of y are increased by 1 and the new address is transferred to P.

NOTE:

This instruction together with the jump indirect instruction provides the means needed for jumping to and from subroutines.

32 LOAD B REGISTER (LB)

Operation: $(y) \rightarrow B$

Execution Time: 1.50 usec.

$y = U_P$ or $U_{SR} + U_{11-0}$

This instruction transfers the contents of y to B specified by IRP. The full 18 bits of y are transferred to B.

The contents of y remain unchanged and in y. $(y)_f = (y)_i$

33 LOAD B REGISTER (LB*)

Operation: $(y + (B)) \rightarrow B$

Execution Time: 1.50 usec.

$y = U_P$ or $U_{SR} + U_{11-0}$

This instruction transfers the contents of y + (B) to B specified by IRP. The full 18 bits of y + (B) are transferred to B.

The contents of y remain unchanged and in y. $(y + (B))_f = (y + (B))_i$

34 JUMP (J)

Operation: $y \rightarrow P_{11-0}$

Execution Time: 0.75 usec.

$y = U_P + U_{11-0}$

This instruction passes program control unconditionally to the location specified by y.

Since only the word address is specified by y and the storage segment address is specified by P_{16-12}, program control remains within the current storage segment.

Example:

$P_{16-12} = 03_8$ and $y = 6712_8$

When the instruction is executed, $P = 036712_8$, and control passes to location 036712.

35 JUMP (J*)

Operation: $y + (B) \rightarrow P_{11-0}$

Execution Time: 0.75 usec.

$y = U_P + U_{11-0}$

This instruction passes program control unconditionally to the location specified by y + (B).

Since the word address is specified by y + (B), the storage segment address specified by P_{16-12} could be modified causing program control to pass to a new location in another storage segment.

36 LOAD B REGISTER WITH "KONSTANT" (LBK)

Operation: $y \rightarrow B$

Execution Time: 0.75 usec.

y = U (sign extended to 18 bits)

This instruction transfers the contents of y to B specified by the index register pointer (IRP). The contents of y is the low-order 12 bits of this instruction U_{11-0} extended to 18 bits by the repetition of bit 11 in bit positions 17 through 12.

Example:

$U_{11-0} = 7701_8$
$(B)_i$ = any value
$(B)_f = 777701_8$

NOTE:

U_{11-0} is the 12-bit number contained within the instruction; it does not refer to an address.

37 LOAD B REGISTER WITH "KONSTANT" (LBK*)

Operation: $y + (B) \rightarrow B$

Execution Time: 0.75 usec.

y = U (sign extended to 18 bits)

This instruction transfers the contents of y ± (B) to B specified by IRP. The contents of y are the low-order 12 bits of this instruction, U_{11-0}, extended to 18 bits by the repetition of bit 11 in bit positions 17 through 12.

The effect of this instruction is to change the contents of B by incrementally increasing or decreasing B.

NOTE:

U_{11-0} is the 12-bit number contained within the instruction; it does not refer to an address.

40 CLEAR Y (CY)

Operation: $0 \rightarrow y$

Execution Time: 1.50 usec.

$y = U_P$ or $U_{SR} + U_{11-0}$

This instruction stores an 18-bit word of 0's at storage address y.

41 CLEAR Y (CY*)

Operation: $0 \rightarrow y + (B)$

Execution Time: 1.50 usec.

$y = U_P$ or $U_{SR} + U_{11-0}$

This instruction stores an 18-bit word of 0's at storage address y + (B).

42 STORE B REGISTER (SB)

Operation: $(B) \rightarrow y$

Execution Time: 1.50 usec.

$y = U_P$ or $U_{SR} + U_{11-0}$

This instruction transfers the contents of B, specified by IRP, to the storage address y.

The contents of B, specified by IRP, remain unchanged and in B. $(B)_f = (B)_i$

43 STORE B REGISTER (SB*)

Operation: $(B) \rightarrow y + (B)$

Execution Time: 1.50 usec.

$y = U_P$ or $U_{SR} + U_{11-0}$

This instruction transfers the contents of B, specified by IRP, to the storage address y + (B).

The contents of B, specified by IRP, remain unchanged and in B. $(B)_f = (B)_i$

44 STORE AL (SL)

Operation: $(AL) \rightarrow y$

Execution Time: 1.50 usec.

$y = U_P$ or $U_{SR} + U_{11\text{-}0}$

This instruction transfers the contents of AL to the storage address y. The contents of AL remain unchanged and in AL. $(AL)_f = (AL)_i$

45 STORE AL (SL*)

Operation: $(AL) \rightarrow y + (B)$

Execution Time: 1.50 usec.

$y = U_P$ or $U_{SR} + U_{11\text{-}0}$

This instruction transfers the contents of AL to the storage address y + (B). The contents of AL remain unchanged and in AL. $(AL)_f = (AL)_i$

46 STORE AU (SU)

Operation: $(AU) \rightarrow y$

Execution Time: 1.50 usec.

$y = U_P$ or $U_{SR} + U_{11\text{-}0}$

This instruction transfers the contents of AU to the storage address y. The contents of AU remain unchanged and in AU. $(AU)_f = (AU)_i$

47 STORE AU (SU*)

Operation: $(AU) \rightarrow y + (B)$

Execution Time: 1.50 usec.

$y = U_P$ or $U_{SR} + U_{11\text{-}0}$

This instruction transfers the contents of AU to the storage address y + (B).

The contents of AU remain unchanged and in AU. $(AU)_f = (AU)_i$

51 INCLUSIVE OR (OR)

Operation: $[(AL)$ **OR** $(y)] \rightarrow AL$

Execution Time: 1.50 usec.

$y = U_P + U_{11\text{-}0}$

Each bit in y is logically added to corresponding bits in AL and the 18 independent logical sums are placed in AL. This is a bit-by-bit INCLUSIVE OR. For each bit in y that equals 1, set the corresponding bit in AL to 1. For each bit that equals 0, the corresponding bit in AL is left as it is.

The contents of y remain unchanged and in y. $(y)_f = (y)_i$

Example:

$(AL)_i = 123456_8$
$(y) \quad = 000077_8$
$(AL)_f = 123477_8$

NOTES:

- The INCLUSIVE OR function is defined in the following table:

(y)	0	0	1	1
(AL)	0	1	0	1
LOGICAL SUM	0	1	1	1

- This instruction is sometimes called selective set.

52 AND (AND)

Operation: [(AL) AND (y)] → AL

Execution Time: 1.50 usec.

$y = U_P + U_{11-0}$

Each bit in y is logically multiplied by corresponding bits in AL and the 18 independent logical products are placed in AL. This is a bit-by-bit AND. For each bit in y that equals 0, clear the corresponding bit in AL to 0. For each bit in y that equals 1, the corresponding bit in AL is left as it is.

The contents of y remain unchanged and in y. $(y)_f = (y)_i$

Example:

$(AL)_i = 123456$
$(y) \quad = 707070$
$(AL)_f = 103050$

NOTES:

- The AND function is defined in the following table:

(y)	0	0	1	1
(AL)	0	1	0	1
LOGICAL PRODUCT	0	0	0	1

- This instruction is sometimes called selective clear.

53 EXCLUSIVE OR (XOR)

Operation: (AL) XOR (y) → AL

Execution Time: 1.50 usec.

$y = U_P + U_{11-0}$

Each bit in y is logically subtracted from corresponding bits in AL and the 18 independent logical differences are placed in AL. This is a bit-by-bit EXCLUSIVE OR. For each bit in y that equals 1, complement the corresponding bit in AL. For each bit in y that equals 0, the corresponding bit in AL is left as it is.

The contents of y remain unchanged and in y. $(y)_f = (y)_i$

Example:

$(AL)_i$ = 123456
(y) = 070007
$(AL)_f$ = 153451

NOTES:

- The EXCLUSIVE OR function is defined in the following table:

(y)	0	0	1	1
(AL)	0	1	0	1
LOGICAL DIFFERENCE	0	1	1	0

- The instruction is sometimes called selective complement.

54 ENABLE INTERRUPTS AND JUMP INDIRECT (EJI)

Operation: (y) → the P register, and remove interrupt lockout

Execution Time: 1.50 usec.

$y = U_P + U_{11-0}$

This instruction removes interrupt lockout, enables interrupts and passes program control to the address which is specified by the contents of y.

NOTES:

- Interrupt lockout is set by all interrupts received from the IOM.
- An application of this instruction is the termination of a subroutine activated by an interrupt.
- This instruction gives the same result as executing the two instructions, clear interrupt lockout (f = 5030) and jump indirect (f = 55), in succession.
- Interrupts are inhibited for one instruction time following the execution of this instruction.

55 JUMP INDIRECT (JI)

Operation: $(y) \rightarrow P$

Execution Time: 1.50 usec.

$y = U_P + U_{11-0}$

This instruction passes program control unconditionally to the location specified by the contents of y.

56 TEST B REGISTER FOR EQUALITY (TB)

Operation: IF $(B) = (y)$; SKIP NI, $[(P) + 2 \rightarrow P]$
IF $(B) \neq (y)$; ADVANCE B BY ONE $[(B) + 1 \rightarrow B]$
EXECUTE NI $[(P) + 1 \rightarrow P]$

Execution Time: 2.50 usec.

$y = U_P + U_{11-0}$

This instruction compares the contents of B, specified by IRP, with the contents of y. If they are equal, the next instruction is skipped. If $(B) = (y)$, then $(P) + 2 \rightarrow P$. If they are not equal, the contents of B are incremented by 1 and the computer executes the next instruction. If $(B) \neq (y)$, then $(B) + 1 \rightarrow B$ and $(P) + 1 \rightarrow P$.

57 TEST ANY LOCATION FOR ZERO (TZ)

Operation: IF $(y) = 0$, SKIP NI, $[(P) + 2 \rightarrow P]$
IF $(y) \neq 0$, DECREMENT (y) BY ONE $[(y) - 1 \rightarrow y]$

EXECUTE NI, $[(P) + 1 \rightarrow P]$

Execution Time: 2.25 usec.

$y = U_P + U_{11-0}$

If the contents of y are 0, the next instruction is skipped. If $(y) = 0$, then $(P) + 2 \rightarrow P$. If they are not 0, they are decremented by 1 and the processor executes the next instruction. If $(y) \neq 0$, then $(y) - 1 \rightarrow y$ and $(P) + 1 \rightarrow P$.

60 JUMP ON AU ZERO (JUZ) (Compare designator not set)

Operation: IF $(AU) = +0,\ y \rightarrow P$
IF $(AU) \neq +0,\ (P) + 1 \rightarrow P$

Execution Time: 0.75 usec.

$y = U_P + U_{11\text{-}0}$

The COMPARE stage of the compare designator is not set.

If the contents of AU equals positive 0, program control passes to the location specified by y. If $(AU) = +0$, then $y \rightarrow P$.

If the contents of AU does not equal positive 0, the processor executes the next instruction. If $(AU) \neq +0$, then $(P) + 1 \rightarrow P$.

NOTE:

Negative 0 acts as not 0.

60 JUMP ON EQUAL (JE) (Compare designator set)

Operation: IF $(AL) = M,\ y \rightarrow P$
IF $(AL) \neq M,\ (P) + 1 \rightarrow P$
IF $[(AL)$ AND $(AU)] = M,\ y \rightarrow P$
IF $[(AL)$ AND $(AU)] \neq M,\ (P) + 1 \rightarrow P$

Execution Time: 0.75 usec.

$y = U_P + U_{11\text{-}0}$

The COMPARE stage of the compare designator is set.

If the EQUAL stage of the compare designator is set, program control passes to the location specified by y.

IF $(AL) = M$, then $y \rightarrow P$
IF (AL) AND $(AU)] = M$, then $y \rightarrow P$

If the EQUAL stage of the compare designator is not set, the next instruction is executed.

IF $(AL) \neq M$, then $(P) + 1 \rightarrow P$
IF $[(AL)$ AND $(AU) \neq M$, then $(P) + 1 \rightarrow P$

NOTES:

- Negative 0 acts as not 0.
- Execution of this instruction does not clear the compare designator.

61 JUMP ON AL ZERO (JLZ) (Compare designator not set)

Operation: $(AL) = +0,\ y \rightarrow P$
$(AL) \neq +0,\ (P) + 1 \rightarrow P$

Execution Time: 0.75 usec.

$y = U_P + U_{11\text{-}0}$

The COMPARE stage of the compare designator is not set.

If the contents of AL equal positive 0, program control passes to the location specified by y. IF $(AL) = +0$, then $y \rightarrow P$

If the contents of AL does not equal positive 0 (contains any 1 bits) the processor executes the next instruction. IF $(AL) \neq 0$, then $(P) + 1 \rightarrow P$

NOTE:

Negative 0 acts as not 0.

61 JUMP ON EQUAL (JE) (Compare designator set)

Operation: IF $(AL) = M,\ y \rightarrow P$
IF $(AL) \neq M,\ (P) + 1 \rightarrow P$
IF $[(AL)$ AND $(AU)] = M,\ y \rightarrow P$
IF $[(AL)$ AND $(AU)] \neq M,\ (P) + 1 \rightarrow P$

Execution Time: 0.75 usec.

$y = U_P + U_{11\text{-}0}$

The COMPARE stage of the compare designator is set.

If the EQUAL stage of the compare designator is set, program control passes to the location specified by y.

IF $(AL) = M$, THEN $y \rightarrow P$
IF $[(AL)$ AND $(AU)] = M$, then $y \rightarrow P$

If the EQUAL stage of the compare designator is not set, the processor executes the next instruction.

IF $(AL) \neq M$, then $(P) + 1 \rightarrow P$
IF $[(AL)$ AND $(AU)] \neq M$, then $(P) + 1 \rightarrow P$

NOTE:

Execution of this instruction does not clear the compare designator.

62 JUMP ON AU NONZERO (JUNZ) (Compare designator not set)

Operation: IF $(AU) \neq +0$, $y \rightarrow P$
IF $(AU) = +0$, $(P) + 1 \rightarrow P$

Execution Time: 0.75 usec.

$y = U_P + U_{11\text{-}0}$

The COMPARE stage of the compare designator is not set.

If the contents of AU does not equal positive 0 (contains any 1 bits) program control passes to the location specified by y. IF $(AU) \neq +0$, then $y \rightarrow P$.

If the contents of AU equals positive 0, the processor executes the next instruction. IF $(AU) = +0$, then $(P) + 1 \rightarrow P$

NOTE:

Negative 0 acts as not 0.

62 JUMP ON NOT EQUAL (JNE) (Compare designator set)

Operation: IF $(AL) \neq M$, $y \rightarrow P$
IF $(AL) = M$, $(P) + 1 \rightarrow P$
IF [(AL) AND (AU)] $\neq$ M, $y \rightarrow P$
IF [(AL) AND (AU)] = M, $(P) + 1 \rightarrow P$

Execution Time: 0.75 usec.

$y = U_P + U_{11\text{-}0}$

The COMPARE stage of the compare designator is set.

If the EQUAL stage of the compare designator is not set, the processor passes control to the location specified by y.

IF $(AL) \neq M$, then $y \rightarrow P$
IF [(AL) AND (AU)] $\neq$ M, then $y \rightarrow P$

If the EQUAL stage of the compare designator is set, the processor executes the next instruction.

IF $(AL) = M$, then $(P) + 1 \rightarrow P$
IF [(AL) AND (AU)] = M, then $(P) + 1 \rightarrow P$

NOTE:

Execution of this instruction does not clear the compare designator.

63 JUMP ON AL NONZERO (JLNZ) (Compare designator not set)

Operation: IF $(AL) \neq +0,\ y \rightarrow P$
IF $(AL) = +0,\ (P) + 1 \rightarrow P$

Execution Time: 0.75 usec.

$y = U_P + U_{11-0}$

The COMPARE stage of the compare designator is not set.

If the contents of AL does not equal positive 0, program control passes to the location specified by y. IF $(AL) \neq +0$, then $y \rightarrow P$

If the contents of AL equals positive 0, the processor executes the next instruction. IF $(AL) = +0$, then $(P) + 1 \rightarrow P$

NOTE:

Negative 0 acts as not 0.

63 JUMP ON NOT EQUAL (JNE) (Compare designator set)

Operation: IF $(AL) \neq M,\ y \rightarrow P$
IF $(AL) = M,\ (P) + 1 \rightarrow P$
IF $[(AL)$ **AND** $(AU) \neq M,\ y \rightarrow P$
IF $[(AL)$ **AND** $(AU)] = M,\ (P) + 1 \rightarrow P$

Execution Time: 0.75 usec.

$y = U_P + U_{11-0}$

The COMPARE stage of the compare designator is set.

If the EQUAL stage of the compare designator is not set, the processor passes control to the location specified by y.

IF $(AL) \neq M$, then $y \rightarrow P$
IF $[(AL)$ **AND** $(AU)] \neq M$, then $y \rightarrow P$

If the EQUAL stage of the compare designator is set, the processor executes the next instruction.

IF $(AL) = M$, then $(P) + 1 \rightarrow P$
IF $[(AL)$ **AND** $(AU)] = M$, then $(P) + 1 \rightarrow P$

NOTE:

Execution of this instruction does not clear the compare designator.

64 JUMP ON AU POSITIVE (JUP) (Compare designator not set)

Operation: IF (AU) POSITIVE, $y \rightarrow P$
IF (AU) NOT POSITIVE, $(P) + 1 \rightarrow P$

Execution Time: 0.75 usec.

$y = U_P + U_{11\text{-}0}$

The COMPARE stage of the compare designator is not set.

If the sign of AU is positive, program control passes to the location specified by y. IF $(AU_{17}) = 0$, then $y \rightarrow P$

If the sign of AU is negative, the processor executes the next instruction. IF $(AU_{17}) = 1$, then $(P) + 1 \rightarrow P$

64 JUMP ON NOT LESS (JNLS) (Compare designator set)

Operation: IF $(AL) \geq M$, $y \rightarrow P$
IF $(AL) < M$, $(P) + 1 \rightarrow P$
IF $[(AL)$ AND $(AU)] \geq M$, $y \rightarrow P$
IF $[(AL)$ AND $(AU)] < M$, $(P) + 1 \rightarrow P$

Execution Time: 0.75 usec.

$y = U_P + U_{11\text{-}0}$

The COMPARE stage of the compare designator is set.

If the LESS THAN stage of the compare designator is not set, program control passes to the location specified by y.

IF $(AL) \geq M$, then $y \rightarrow P$
IF $[(AL)$ AND $(AU)] \geq M$, then $y \rightarrow P$

If the LESS THAN stage of the compare designator is set, the processor executes the next instruction.

IF $(AL) < M$, $(P) + 1 \rightarrow P$
IF $[(AL)$ AND $(AU)] < M$, $(P) + 1 \rightarrow P$

NOTE:

Execution of this instruction does not clear the compare designator.

65 JUMP ON AL POSITIVE (JLP) (Compare designator not set)

Operation: IF (AL) POSITIVE, $y \rightarrow P$
IF (AL) NEGATIVE, $(P) + 1 \rightarrow P$

Execution Time: 0.75 usec.

$y = U_P + U_{11-0}$

The COMPARE stage of the compare designator is not set.

If the sign of AL is positive, program control passes to the location specified by y. IF $(AL_{17}) = 0$, then $y \rightarrow P$

If the sign of AL is negative, the processor executes the next instruction. IF $(AL_{17}) = 1$, then $(P) + 1 \rightarrow P$

65 JUMP ON NOT LESS (JNLS) (Compare designator set)

Operation: IF $(AL) \geq M$, $y \rightarrow P$
IF $(AL) < M$, $(P) + 1 \rightarrow P$
IF [(AL) AND (AU)] $\geq M$, $y \rightarrow P$
IF [(AU) AND (AU)] $< M$, $(P) + 1 \rightarrow P$

Execution Time: 0.75 usec.

$y = U_P + U_{11-0}$

The COMPARE stage of the compare designator is set.

If the LESS THAN stage of the compare designator is not set, program control passes to the location specified by y.

IF $(AL) \geq M$, then $y \rightarrow P$
IF [(AL) AND (AU)] $\geq M$, then $y \rightarrow P$

If the LESS THAN stage of the compare designator is set, the processor executes the next instruction.

IF $(AL) < M$, $(P) + 1 \rightarrow P$
IF [(AL) AND (AU)] $> M$, $(P) + 1 \rightarrow P$

NOTE:

Execution of this instruction does not clear the compare designator.

66 JUMP ON AU NEGATIVE (JUN) (Compare designator not set)

Operation: IF (AU) NEGATIVE, $J \rightarrow P$
IF (AU) POSITIVE, $(P) + 1 \rightarrow P$

Execution Time: 0.75 usec.

$y = U_P + U_{11\text{-}0}$

The COMPARE stage of the compare designator is not set.

If the sign of AU is negative, program control passes to the location specified by y.

IF $(AU_{17}) = 1$, then $y \rightarrow P$

If the sign of AU is positive, the processor executes the next instruction.
IF $(AU_{17}) = 0$, then $(P) + 1 \rightarrow P$

66 JUMP ON LESS (JLS) (Compare designator set)

Operation: IF $(AL) < M$, $y \rightarrow P$
IF $(AL) \geq M$, $(P) + 1 \rightarrow P$
IF [(AL) AND (AU)] $< M$, $y \rightarrow P$
IF [(AL) AND (AU)] $\geq M$, $(P) + 1 \rightarrow P$

Execution Time: 0.75 usec.

$y = U_P + U_{11\text{-}0}$

The COMPARE stage of the compare designator is set.

If the LESS THAN stage of the compare designator is set, program control passes to the location specified in y.

IF $(AL) < M$, then $y \rightarrow P$
IF [(AL) AND (AU)] $< M$, then $y \rightarrow P$

If the LESS THAN stage of the compare designator is not set, the processor executes the next instruction.

IF $(AL) \geq M$, then $(P) + 1 \rightarrow P$
IF [(AL) AND (AU)] $\geq M$, then $(P) + 1 \rightarrow P$

NOTE:

Execution of this instruction does not clear the compare designator.

67 JUMP ON AL NEGATIVE (JLN) (Compare designator not set)

Operation: IF (AL) NEGATIVE, y $\rightarrow$ P
IF (AL) POSITIVE, (P) + 1 $\rightarrow$ P

Execution Time: 0.75 usec.

$y = U_P + U_{11-0}$

The COMPARE stage of the compare designator is not set.

If the sign of AL is negative, program control passes to the location specified by y.

If $(AL_{17}) = 1$, then y $\rightarrow$ P

If the sign of AL is positive, the processor executes the next instruction. IF (AL) = 0, then (P) + 1 $\rightarrow$ P

67 JUMP ON LESS (JLS) (Compare designator set)

Operation: IF (AL) < M, y $\rightarrow$ P
IF (AL) $\geq$ M, (P) + 1 $\rightarrow$ P
IF [(AL) AND (AU)] < M, y $\rightarrow$ P
IF [(AL) AND (AU)] $\geq$ M, (P) + 1 $\rightarrow$ P

Execution Time: 0.75 usec.

$y = U_P + U_{11-0}$

The COMPARE stage of the compare designator is set.

If the LESS THAN stage of the compare designator is set, program control passes to the location specified by y.

IF (AL) < M, then y $\rightarrow$ P
IF [(AL) AND (AU)] < M, then y $\rightarrow$ P

If the LESS THAN stage of the compare designator is not set, the processor executes the next instruction.

IF (AL) $\geq$ M, then (P) + 1 $\rightarrow$ P
IF [(AL) AND (AU)] $\geq$ M, then (P) + 1 $\rightarrow$ P

NOTE:

Execution of this instruction does not clear the compare designator.

70 LOAD AL WITH "KONSTANT" (LLK)

Operation: $y \rightarrow AL$

Execution Time: 1.00 usec.

$y = U$ (sign extended to 18 bits)

The contents of y are the lower-order 12 bits of this instruction extended to 18 bits by the repetition of bit 11 in bit positions 17 through 12. This expanded 18-bit number is placed in AL.

Examples:

70 0001_8, $y = 0001_8$, LOAD AL WITH "KONSTANT" + 1

$(AL)_i$ = any value
$(AL)_f = 000001_8$
70 7775_8, $y = 7775_8$, LOAD AL WITH "KONSTANT" – 1
$(AL)_i$ = any value
$(AL)_f = 777775_8$

NOTES:

- The LOAD AL WITH "KONSTANT" instruction itself remains unchanged by the operation.
- U is the 12-bit number contained within the instruction; it does not refer to an address.
- The constant, U, may range in value from -3777_8 to $+3777_8$.

71 ADD "KONSTANT" TO AL (ALK)

Operation: $(AL) + y \rightarrow AL$

Execution Time: 1.00 usec.

y = U (sign extended to 18 bits)

The contents of y are the lower-order 12 bits of this instruction, extended to 18 bits by the repetition of bit 11 in bit positions 17 through 12. This 18-bit number is then added to the contents of AL and the resultant, SUM, is placed in AL.

Examples:

71 0002_8, y = 0002_8, ADD "KONSTANT" + 2 TO AL
$(AL)_i = 057777_8$
$(AL)_f = 060001_8$

71 7775_8, y = 7775_8, ADD "KONSTANT" – 2 TO AL
$(AL)_i = 067055_8$
$(AL)_f = 067053_8$

NOTES:

- The ADD "KONSTANT" TO AL instruction itself remains unchanged by the operation.
- U is the 12-bit number contained within the instruction; it does not refer to an address.
- The constant, U, may range in value from -3777_8 to $+3777_8$.
- If the contents of AL is negative 0 and y is negative 0, the result of the addition is negative 0.

  $(AL)_f$ = 1's if $(AL)_i$ = 1's and y = 1's
- The results of addition involving all other possible combinations of positive and negative 0 are positive 0.
- If the magnitude of the resultant is too large for AL to hold, the result is incorrect and the overflow designator is set. The state of the overflow designator is tested by either the SKIP ON OVERFLOW instruction (f = 5052) or the SKIP ON NO OVERFLOW instruction (f = 5053). The execution of either of these two instructions clears the overflow designator.

72 STORE INDEX REGISTER (SIR)

Operation: $IRP_{3\text{-}0} \rightarrow y_{3\text{-}0}$
$0\text{'s} \rightarrow y_{5\text{-}4}$

Execution Time: 3.00 usec.

$y = U_P + U_{11\text{-}0}$

This instruction replaces the six low-order bits of the contents of y with a six-bit value in which the contents of $IRP_{3\text{-}0}$ replaces the contents of $y_{3\text{-}0}$ and zeros replace the contents of $y_{5\text{-}4}$. Bits 17 through 6 of the contents of y remain unchanged. The resultant is stored at storage location y.

NOTES:

- If the contents of IRP equals 0, bit 3 of the contents of y is set. If the contents of IRP does not equal 0, bit 3 of the contents of y is cleared. That is, IRP points to storage address 10_8 when loaded with 00_8.

  IF $(IRP) = 0\ (y_3) = 1$
  IF $(IRP) \neq 0\ (y_3) = 0$

- Since this instruction effects a partial transfer, the 12 high-order bits of y remain unchanged.

73 JUMP IF B REGISTER NONZERO (JBNZ)

Operation: IF $(B) +0,\ (B) - 1 \rightarrow B$ and $y \rightarrow P$
IF $(B) +0,\ (P) + 1 \rightarrow P$

Execution Time: 1.75 usec.

$y = U_P + U_{11\text{-}0}$

If the contents of B, specified by IRP, are not positive 0, the contents of B are decremented by 1 and program control passes to the location specified by y. If the contents of B, specified by IRP, are positive 0, the processor executes the next instruction.

IF $(B) +0$, then $(B) - 1 \rightarrow B$ and $y \rightarrow P$
IF $(B) +0$, then $(P) + 1 \rightarrow P$

NOTES:

- Negative 0 acts as not 0.

- Since B is a one's complement number and can take values less than zero, the B JUMP is effective for program loops only when the contents of B is initially positive.

74 STORE ADDRESS OF AL (SAD)

Operation: $(AL_{11-0}) \rightarrow y_{11-0}$

Execution Time: 3.00 usec.

$y = U_P + U_{11-0}$

The low-order 12 bits of the contents of AL, (AL_{11-0}), replace the corresponding low-order 12 bits of the contents of y, (y_{11-0}). The high-order six bits of the contents of y (y_{17-12}) remain unchanged.

The contents of AL remain unchanged and in AL.

Example:

$(AL)_i = 762504_8$
$(y)_i = 567777_8$
$(y)_f = 562504_8$

NOTE:

Since this instruction effects a partial transfer, the six high-order bits of y remain unchanged.

75 STORE SPECIAL REGISTER (SSR)

Operation: $(SR_{5-0}) \rightarrow y_{5-0}$

Execution Time: 3.00 usec.

$y = U_P + U_{11-0}$

The contents of the special register replace the 6 low-order bits of the contents of y (y_{5-0}). Bit 4 of the special register, SR_4, is cleared to 0. The contents of $SR_{3-0,5}$ bits 0 through 3 and bit 5, and the contents of y_{17-6} bits 17 through 6, remain unchanged by the operation.

NOTES:

- Since the instruction effects a partial transfer, bits 17 through 6 of the contents of y (y_{17-6}) remain unchanged.
- This instruction deactivates the special register as the control bit, bit 4, is cleared.

76 STORE LOCATION AND JUMP (SLJ)

Operation: $(P) + 1 \rightarrow y$ and $y + 1 \rightarrow P$

Execution Time: 2.00 usec.

$y = U_P + U_{11\text{-}0}$

The address of the next instruction in storage replaces the contents of the location specified by y; that is, the current program address plus 1 is stored in y. Program control passes to the location following the location specified by y; that is, jump to y plus 1.

NOTES:

- This instruction transfers a full 18-bit word to y.
- The lower 17 bits are (P) + 1; the upper bit is set to 0.

6.4. TYPE III INSTRUCTIONS

The following are type III instructions. Each requires a function code of 50 and a minor function code in the range of 00_8 through 77_8. The 50 function code identifies the instruction as type III; the minor function code determines the operation to be performed.

6.4.1. Type III-b Instructions

Most of the type III-b instructions are the optional floating-point instructions. In processors not equipped with this feature, floating-point commands are considered as faults and generate a supervisor call interrupt.

5002 FLOATING-POINT ADD (FA) and (FA*)

Operation: (FA)
$[(A) + (y-1,y)] \rightarrow A$
(FA*)
$[(A) + (y-1 + (B), y + (B))] \rightarrow A$

Execution Time: (4.35 + number of shifts/8) usec.

$y = U_P$ or $U_{SR} + U_{11\text{-}0}$

This instruction causes the signed floating-point number contained in the main storage addresses specified by y−1 (most significant half) and y (least significant half) to be added to the signed floating-point number contained in the A register. The sign is indicated by the most significant bit of y−1. The characteristics are compared and the fixed-point part and exponent in the floating-point number with the smallest exponent are adjusted until the two exponents are the same. The fixed-point parts are added, the sum is normalized, and the result is placed in the A register in the floating-point format. AU_{17} contains the resultant sign. $AU_{16\text{-}9}$ contains the resultant exponent and $AU_{8\text{-}0}$ and $AL_{17\text{-}0}$ contain the resultant fixed-point part.

NOTES:

- If the resultant exponent is less than zero and the resultant fixed-point part is nonzero, the operation is completed by normalizing the fixed-point part and decrementing the exponent past zero, packing the result in A, and causing an underflow interrupt to location 34_8.

- If the resultant exponent is greater than 377_8 and the resultant fixed-point part is nonzero, the operation is completed by normalizing the fixed-point part (shift right one place), incrementing and truncating the exponent (which results in a zero exponent), packing the result in A, and causing an overflow interrupt to location 35_8.

- If the resultant fixed-point part is a plus or minus 0, a plus 0 is placed in the A register and no interrupt is generated.

5003 FLOATING-POINT SUBTRACT (FS) AND (FS*)

Operation: (FS)
$[(A) - (y-1,y)] \rightarrow A$
(FS*)
$[(A) - (y-1 + (B), y + (B))] \rightarrow A$

Execution Time: (4.35 + number of shifts/8) usec.

$y = U_P$ or $U_{SR} + U_{11\text{-}0}$

This instruction causes the signed floating-point number contained in the main storage addresses specified by y−1 (most significant half) and y (least significant half) to be subtracted from the signed floating-point number contained in the A register. The sign is indicated by the most significant bit of y−1. The exponents are compared and the fixed-point part and exponent in the floating-point number with the smallest exponent are adjusted until the two exponents are the same. After subtraction, the difference is normalized and the result is contained in the A register in the floating-point format.

AU_{17} contains the resultant sign. $AU_{16\text{-}9}$ contains the resultant exponent and $AU_{8\text{-}0}$ and $AL_{17\text{-}0}$ contain the resultant fixed-point part.

NOTES:

- If the resultant exponent is less than zero and the resultant fixed-point part is nonzero, the operation is completed by normalizing the exponent and decrementing the fixed-point part past zero, packing the result in A, and causing an underflow interrupt to location 34_8.

- If the resultant exponent is greater than 377_8 and the resultant fixed-point part is nonzero, the operation is completed by normalizing the fixed-point part (shift right one place), incrementing and truncating the exponent (which results in a zero exponent), packing the result in A, and causing an overflow interrupt to location 35_8.

- If the resultant fixed-point part is a plus or minus 0, a plus 0 is placed in the A register and no interrupt is generated.

5004 FLOATING-POINT MULTIPLY (FM) and (FM*)

Operation: (FM)
$[(A) \times (y-1,y)] \rightarrow A$
(FM*)
$[(A) \times (y-1 + (B), y + (B))] \rightarrow A$

Execution Time: 12.00 usec.

$y = U_P \text{ or } U_{SR} + U_{11\text{-}0}$

This instruction causes the signed floating-point number contained in the A register to be multiplied by the contents of the signed floating-point number contained in the main storage address specified by y – 1 (most significant half) and y (least significant half), with the product contained in the A register in the floating-point format. AU_{17} contains the resultant sign. $AU_{16\text{-}9}$ contains the resultant exponent and $AU_{8\text{-}0}$ and $AL_{17\text{-}0}$ contain the resultant fixed-point part.

NOTES:

- If the resultant exponent is less than zero, the operation is completed by placing the resulting exponent (which is truncated to 8 bits) and the normalized fixed-point part (shifted zero or one place left, since operands are assumed to be normalized) in A, then causing an interrupt to location 34_8.
- If the resultant exponent is greater than 377_8, the operation is completed by placing the resulting exponent (truncated to 8 bits) and the normalized fixed-point part in A, then causing an interrupt to location 35_8.

5005 FLOATING-POINT DIVIDE (FD) AND (FD*)

Operation: (FD)
$[(A) + (y-1,y)] \rightarrow A$
(FD*)
$[(A) + (y-1 + (B), y + (B))] \rightarrow A$

Execution Time: 12.00 usec.

$y = U_P \text{ or } U_{SR} + U_{11\text{-}0}$

This instruction causes the signed floating-point number contained in the A register to be divided by the contents of the signed floating-point number contained in the main storage addresses specified by y–1 (most significant half) and y (least significant half), with the quotient contained in the A register in the floating-point format. The remainder is not saved. AU_{17} contains the resultant sign. $AU_{16\text{-}9}$ contains the resultant exponent and $AU_{8\text{-}0}$ and $AL_{17\text{-}0}$ contain the resultant fixed-point part.

NOTES:

- If division is attempted with an unnormalized divisor or a divisor of plus or minus 0, the operation is suppressed (contents of A is unchanged), the overflow designator is set, and an exponent overflow interrupt occurs to location 35_8.
- If the resultant exponent is less than 0, the operation is completed by placing the resulting exponent (truncated to 8 bits) and the normalized fixed-point part in A, then causing an interrupt to location 34_8.
- If the resultant exponent is greater than 377_8, the operation is completed by placing the resulting exponent (truncated to eight bits) and the normalized fixed-point part (right shift of zero or one place) in A, then causing an interrupt to location 35_8.

5006 FLOATING-POINT PACK (FP) AND (FP*)

Operation: (FP)

$(A_{35}) \rightarrow A_{35\text{-}27}$

Normalized $(A_{35\text{-}0}) \rightarrow A_{26\text{-}0}$

$[(Y_{7\text{-}0}) \pm \text{actual shift count}]$ XOR $A_{34\text{-}27} \rightarrow A_{34\text{-}27}$

(FP*) when bit position 12 of the second word = 1,

$\{[y+(B)_{7\text{-}0}] \pm \text{actual shift count}\}$ XOR $A_{34\text{-}27} \rightarrow A_{34\text{-}27}$

Execution Time: (3.5 + number of shifts/8) usec.

$y = U_P \text{ or } U_{SR} + U_{11\text{-}0}$

The contents of the A register (the fixed-point part) is normalized by shifting the contents of A left or right until the most significant bit of the number is in bit position 26. The sign bit, A_{35}, is extended through bit positions 35-27. The contents of bit positions 7 through 0 of the main storage address specified by Y (the exponent part) plus the number of right shifts or minus the number of left shifts necessary for the normalization is exclusively ORed into bit positions 34-27.

Examples:

(1)	(AU) i = 000000	(AL)i = 000001	(y)i = 000233
	(AU) f = 201400	(AL)f = 000000	(y)f = 000233
(2)	(AU) i = 777777	(AL)i = 777773	(y)i = 000233
	(AU) f = 575377	(AL)f = 777777	(y)f = 000233
(3)	(AU)i = 123456	(AL)i = 712345	(y)i = 000233
	(AU)f = 242516	(AL)f = 273451	(y)f = 000233
(4)	(AU)i = 100000	(AL)i = 000000	(y)i = 000000
	(AU)f = 007400	(AL)f = 000000	(y)f = 000000

NOTES:

- If the contents of the A register are initially plus or minus 0, the result is plus 0.

- Overflow and underflow are handled the same as in FA and FS (see notes given with FA and FS instructions), except that a right shift of eight places may cause the exponent to overflow past 0.

- The contents of the operand address are normally 0233_8 (bias $+27_{10}$) for a float operation. For example, to float an integer value given in (AL):

SLA 18	Put sign into (AU)
SRA 18	Restore (AL)
FP (0200+27)	Float

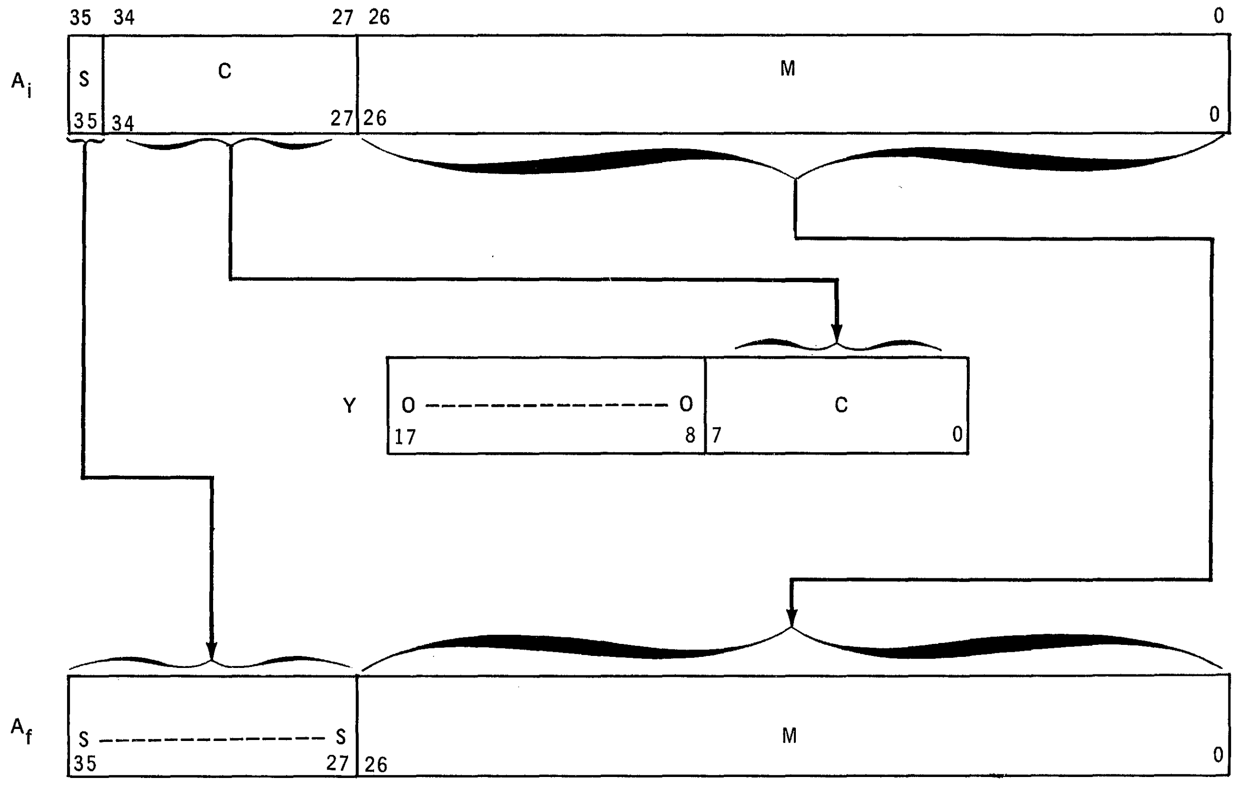
5007 FLOATING-POINT UNPACK (FU) and (FU*)

Operation: (FU)
If $(A_{35}) = 0$, $(A_{34\text{-}27}) \rightarrow y_{7\text{-}0}$
If $(A_{35}) = 1$, $(\overline{A_{34\text{-}27}}) \rightarrow y_{7\text{-}0}$ and 0's $\rightarrow y_{17\text{-}8}(A_{35}) \rightarrow A_{34\text{-}27}$
(FU*) If bit position 12 of the second word = 1,
$(A_{34\text{-}27})$ or $(\overline{A_{34\text{-}27}}) \rightarrow [y + (B)]_{7\text{-}0}$

Execution Time: 3.50 usec.

$y = U_P$ or $U_{SR} + U_{11\text{-}0}$

The contents of the absolute value of the exponent (that is, if S = 1, complement the exponent) in the A register bit positions 34 through 27 are transferred into bit positions 7 through 0 of the main storage address specified by y. If A_{35} is a 1 the exponent is complemented before storing. Zeros are put into $y_{17\text{-}8}$. The content of bit position 35 of the A register is put into bit positions 34 through 27 of the A register. Bit positions 26 through 0 of the A register are unchanged.



5010 READ AND SET (RS) and (RS*)

Operation: $(y) \rightarrow AL$ or $[y + (B)] \rightarrow AL$
$(y_{16\text{-}0}) \rightarrow y_{16\text{-}0}$ and $1 \rightarrow y_{17}$ or $[y + (B)]_{16\text{-}0} \rightarrow [y + (B)]_{16\text{-}0}$ and $1 \rightarrow [y + (B)]_{17}$

Execution Time: 2.50 usec.

$y = U_P$ or $U_{SR} + U_{11\text{-}0}$

This instruction transfers the contents of y, bits 17 through 0, into AL. Then bits 16 through 0 are restored to y, and bit 17 of y is set to 1.

If bit 12 of (P + 1) is set, then the address is B modified.

6.4.2. Type III-a Instructions

5011 LOAD INPUT CHANNEL (LIC) – Privileged

Operation: Load I/O channel K from (P) + 1 and (P) + 2.
Initiate input, (P) + 3 → P.

Execution Time: 5.30 usec minimum.

Execution of this instruction activates the input channel specified by the K portion of the instruction and causes the two succeeding addresses to be stored in the input buffer control word addresses for the designated channel, (P) + 1 = terminal buffer control word and (P) + 2 = present buffer control word. The processor then resumes normal operation by passing program control to the location immediately following the buffer control words, (P) + 3 → P. The contents of the two storage registers following the instruction remain unchanged by the operation.

NOTES:

- On ESI channels, the two words following a load input channel instruction are ignored since buffer control addresses are obtained from the communications line terminal (CLT).
- K must be odd for paired channel, 36-bit operation.

5012 LOAD OUTPUT CHANNEL (LOC) – Privileged

Operation: Load I/O channel K from (P) + 1 and (P) + 2.
Initiate output, (P) + 3 → P.

Execution Time: 5.30 usec minimum.

Execution of this instruction activates the output channel specified by the K portion of the instruction and causes the two succeeding addresses to be stored in the output buffer control word addresses for the designated channel, (P) + 1 = terminal buffer control word and (P) + 2 = present buffer control word. The processor then resumes normal operation by passing program control to the location immediately following the buffer control word, (P) + 3 → P. The contents of the two storage registers following the instruction remain unchanged by the operation.

NOTES:

- On ESI channels, the two words following a load output channel instruction are ignored since buffer control addresses are obtained from the communications line terminal (CLT).
- K must be odd for paired channel, 36-bit operation.

5013 LOAD EXTERNAL FUNCTION CHANNEL (LFC) – Privileged

Operation: Load I/O channel K from (P) + 1 and (P) + 2.
Initiate external function, (P) + 3 → P.

Execution Time: 5.60 usec minimum.

Execution of this instruction activates the input channel specified by the K portion of the instruction and causes the two succeeding addresses to be stored in the input buffer control word addresses for the designated channel, (P) + 1 = terminal buffer control word and (P) + 2 = present buffer control word. The processor then resumes normal operation by passing program control to the location immediately following the buffer control words, (P) + 3 → P. The contents of the 2 storage registers following the instruction remain unchanged by the operation.

NOTES:

- K must be odd for paired channel, 36-bit operation.
- K must be even for channels in ESI mode.

5015 STOP INPUT ON CHANNEL (STIC) – Privileged

Operation: Stop input on channel K.

Execution Time: 2.15 usec minimum.

Execution of this instruction stops all input activity on the channel specified by the K portion of the instruction.

NOTE:

K should be odd for paired, 36-bit channel operation.

5016 STOP OUTPUT ON CHANNEL (STOC) – Privileged

Operation: Stop output or external function on channel K.

Execution Time: 2.15 usec minimum.

Execution of this instruction stops all output or external function activity on the channel specified by the K portion of the instruction.

NOTE:

K should be odd for paired, 36-bit channel operation.

5017 STORE SPECIAL DESIGNATORS (SSD)

Operation: Store the contents of SR and of the borrow and overflow designators into the address specified by (P)+1; (P)+2 → P.

Execution Time: 2.50 usec.

The designator settings and the SR contents will be stored in the following format:

0	B	OV	0	SR
17 12	11	10	9 6	5 0

B is set to 1 if the borrow designator is set; 0 if it is not.
OV is set to 1 if the overflow designator is set; 0 if it is not.

5020 LOAD SPECIAL DESIGNATORS (LSD)

Operation: Load the SR register and set the borrow and overflow designators with the contents of the address specified by (P)+1; (P)+2 → P.

Execution Time: 2.50 usec.

The SR is loaded with bits 5–0 of the word specified at (P)+1. The borrow and overflow designators are set with the values of bit positions 11 and 10 of the word specified by (P)+1.

5021 TEST INPUT CHANNEL (TIC) – Privileged

Operation: If input channel K is idle $(P) + 2 \rightarrow P$
If input channel K is active $(P) + 1 \rightarrow P$

Execution Time: 1.00 usec.

This instruction tests for input activity on the channel specified by the K portion of the instruction. If there is no input activity on channel K, the next instruction is skipped. If there is activity on channel K, the next instruction is executed; $(P) + 1 \rightarrow P$.

NOTE:

K should be the same as in the load input channel instruction, 5011.

5022 TEST OUTPUT CHANNEL (TOC) – Privileged

Operation: If output channel K is idle $(P) + 2 \rightarrow P$
If output channel K is active $(P) + 1 \rightarrow P$

Execution Time: 1.00 usec.

This instruction tests for output activity or external function activity on the channel specified by the K portion of the instruction. If there is no output activity or external function activity on channel K, the next instruction is skipped; $(P) + 2 \rightarrow P$. If there is output activity on channel K, the next instruction is executed; $(P) + 1 \rightarrow P$.

NOTE:

K should be the same as in the load output channel instruction, 5012.

5023 TEST FUNCTION CHANNEL (TFC) – Privileged

Operation: If external function channel K is idle $(P) + 2 \rightarrow P$
If external function channel K is active $(P) + 1 \rightarrow P$

Execution Time: 1.00 usec.

This instruction tests for external function activity on the channel specified by the K portion of the instruction. If there is no external function activity on channel K, the next instruction is skipped; $(P) + 2 \rightarrow P$. If there is external function activity on channel K, the next instruction is executed; $(P) + 1 \rightarrow P$.

NOTE:

K should be the same as in the load external function channel instruction, 5013.

5024 or 5025 WAIT FOR INTERRUPT (WFI) – Privileged

Operation: Stop c/a section, but not I/O transmission until the occurrence of an interrupt.

Execution Time: 1.00 usec.

This instruction stops the main program operation, but lets I/O activity continue normally. When an interrupt of any type occurs, the interrupt is processed, and main program operation is resumed. K is ignored.

5026 NO OPERATION (NOP)

Operation: $(P) + 1 \rightarrow P$

Execution Time: 1.00 usec.

The execution of this instruction increments the contents of P by 1, $(P) + 1 \rightarrow P$. No other operation occurs as a result of this instruction.

5030 or 5031 ALLOW ALL INTERRUPTS (AAI)

Operation: Remove I/O interrupt lockout.

Execution Time: 1.00 usec.

This instruction permits all I/O interrupts to be honored after having been locked out by the prevent all interrupts instruction, 5034 or 5035, or by the occurrence of an interrupt. K is ignored. Interrupts are inhibited for one instruction time following the execution of this instruction.

5034 or 5035 PREVENT ALL INTERRUPTS (PAI)

Operation: Locks out I/O interrupts.

Execution Time: 1.00 usec.

This instruction prevents all I/O interrupts from being honored. K is ignored.

NOTES:

- This instruction stops interrupts from the delta clock and day clock but allows updating of them while preventing all I/O interrupts.
- This instruction has the same effect as the occurrence of an interrupt.

5041 RIGHT SHIFT AU (SRU)

Operation: Shift (AU) right K bit positions.

Execution Time: (1.00 + number of shifts/8) usec.

The contents of AU are shifted to the right by the number of bit positions specified by the K portion of the instruction. The original sign bit of AU, the content of AU_{17}, at the time the shift begins is filled in at the left end of AU. In all cases, this is an end-off shift; the lower-order bits of AU, specified by K, are lost off the right end of AU.

Example:

K = 2 and the contents of AU are positive
$(AU)_i = 370000_8$

First Shift
$(AU) = 174000_8$

Second Shift
$(AU)_f = 076000_8$

K = 2 and the contents of AU are negative
$(AU)_i = 400000_8$

First Shift
$(AU) = 600000_8$

Second Shift
$(AU)_f = 700000_8$

5042 RIGHT SHIFT AL (SRL)

Operation: Shift (AL) right K bit positions.

Execution Time: (1.00 + number of shifts/8) usec.

The contents of AL are shifted to the right by the number of bit positions specified by the K portion of the instruction. The original sign bit of AL, the contents of AL_{17}, at the time the shift begins is filled in at the left end of AL. In all cases, this is an end-off shift; the low-order bits of AL, specified by K, are lost off the right end of AL.

5043 RIGHT SHIFT A (SRA)

Operation: Shift (A) right K bit positions.

Execution Time: (1.00 + number of shifts/8) usec.

The contents of A are shifted to the right by the number of bit positions specified by the K portion of the instruction. The low-order bit of AU, the contents of AU_0, becomes the high-order bit or sign bit of AL, the contents of AL_{17}. The original sign bit of A, the contents of A_{35}, at the time the shift begins is filled in at the left end of A. In all cases, this is an end-off shift; the low-order bits of A, specified by K, are lost off the right end of A.

Example:

K = 2 and the contents of A is positive
$(A)_i = 370000\ 000000_8$

First Shift
$(A) = 174000\ 000000_8$

Second Shift
$(A)_f = 076000\ 000000_8$

K = 2 and the contents of A is negative
$(A)_i = 400000\ 000000_8$

First Shift
$(A) = 600000\ 000000_8$

Second Shift
$(A)_f = 700000\ 000000_8$

5044 SCALE A (SCA)

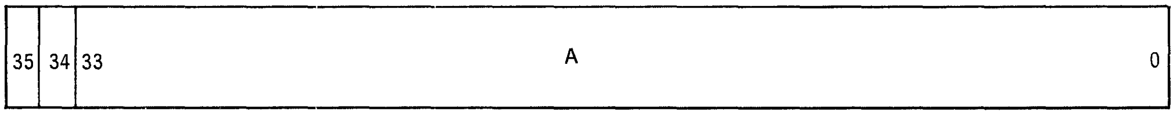
Operation: Shift (A) left circularly by K bit positions or until (A) is normalized; K less the actual shift count (location 000017_8).

Execution Time: (2.00 + number of shifts/8) usec.

If the K portion of the instruction is less than or equal to the shift count needed to normalize the contents of A, the contents of A are shifted left by the number of bit positions specified by K and positive 0 is stored at storage location 000017_8.

If the K portion of the instruction is greater than the shift count needed to normalize the contents of A, the contents of A become normalized and the number of bit positions that the contents of A are actually shifted is subtracted from K and the difference is stored in storage location 000017_8. The contents of A become normalized by shifting the contents of A left until the most significant bit of the number is in bit position 34, A_{34}. In the case of a positive number, the content of A_{34} equals 1, and in the case of a negative number, the content of A_{34} equals 0. The content of A_{35} cannot equal the content of A_{34} for a normalized number.

Example:



K = 7
$(A)_i$ = 170000 000000_8 (positive and not normalized)

First Shift
$(A)_f$ = 360000 000000_8 (positive and normalized)

The processor senses that the contents of A are normalized and stores the quantity K minus the shift count, (000007_8–000001_8) = (000006_8), at storage address 000017_8.

K = 3
$(A)_i$ = 600000 000000_8 (negative and not normalized)

First Shift
$(A)_f$ = 400000 000001_8 (negative and normalized)

When the contents of A is normalized, the quantity K minus the shift count is stored; (000003_8) – (000001_8) = (000002_8), at storage address 000017_8.

K = 1
$(A)_i$ = 070000 000000_8 (positive and not normalized)

First Shift
$(A)_f$ = 160000 000000_8 (positive and not normalized)

When the number of bit positions specified by K have been shifted, the quantity 000000_8 is stored at storage address 000017_8. The contents of A are only partially normalized.

NOTE:

This instruction is useful in the conversion of numbers to a floating-point format.

5045 LEFT SHIFT AU (SLU)

Operation: Shift (AU) left K bit positions.

Execution Time: (1.00 + number of shifts/8) usec.

The contents of AU are shifted to the left by the number of bit positions specified by the K portion of the instruction. The high-order bits that are shifted out through the left end of AU fill in the low-order bit positions of AU. No bits are lost as a result of the operation.

Example:

K = 2
$(AU)_i$ = 300000_8

First Shift
(AU) = 600000_8

Second Shift
$(AU)_f$ = 400001_8

5046 LEFT SHIFT AL (SLL)

Operation: Shift (AL) left K bit positions.

Execution Time: (1.00 + number of shifts/8) usec.

The contents of AL are shifted to the left by the number of bit positions specified by the K portion of the instruction. The high-order bits that are shifted out through the left end of AL fill in the low-order bit positions of AL. No bits are lost as a result of the operation.

5047 LEFT SHIFT A (SLA)

Operation: Shift (A) left K bit positions.

Execution Time: (1.25 + number of shifts/8) usec.

The contents of A are shifted to the left by the number of bit positions specified by the K portion of the instruction. The high-order bits that are shifted out through the left end of A fill in the low-order bit positions of A. No bits are lost as a result of the operation.

Example:

K = 2
$(A)_i = 300000\ 000000_8$

First Shift
$(A) = 600000\ 000000_8$

Second Shift
$(A)_f = 400000\ 000001_8$

5050 TEST KEYS (TK)

Operation: If keys designated by K are set, (P) + 2 → P

Execution Time: 1.00 usec.

There are five skip keys on the UNIVAC 418-III maintenance panel and console which, together with this instruction, permit external control of program branching. Bits 4 through 0 of the K portion of this instruction correspond to skip keys 4 through 0 on the maintenance panel and console. For every bit in $K_{4\text{-}0}$ that is set to 1, the corresponding skip key is examined. If any of the examined keys are set, the next instructions are skipped; (P) + 2 → P. If K equals 0 or if all the examined keys are not set, the next instruction is executed; (P) + 1 → P. If K_5 equals 1, the state of $K_{4\text{-}0}$ is ignored, and the next instruction is skipped; (P) + 2 → P.

Example:

K = 01 (bit 0) skip if skip key 0 is set.
K = 02 (bit 1) skip if skip key 1 is set.
K = 04 (bit 2) skip if skip key 2 is set.
K = 10 (bit 3) skip if skip key 3 is set.
K = 20 (bit 4) skip if skip key 4 is set.
K = 40 (bit 5) skip unconditionally.
K = 03 (bits 1,0) skip if skip key 1 or 0 is set.

NOTE:

All combinations of octal codes 00 through 77 are valid codes for K.

5051 TEST NO BORROW (TNB)

Operation: If borrow designator is not set (P) + 2 → P
If borrow designator is set (P) + 1 → P

Execution Time: 1.00 usec.

This instruction tests the condition of the borrow designator and passes program control accordingly. If a double-length add or subtract required a borrow, the next instruction is skipped; (P) + 2 → P. K is ignored. If a skip does not occur, a correction of the contents of A is needed. The contents of A will be too large by a factor of 1. The correcting instruction is ADD NEGATIVELY TO A. This allows a correcting instruction to be inserted to save program steps.

5052 TEST OVERFLOW (TOF)

Operation: If overflow designator is set (P) + 2 → P
If overflow designator is not set (P) + 1 → P

Execution Time: 1.00 usec.

This instruction tests the condition of the overflow designator and passes program control accordingly. If an overflow condition occurred on an arithmetic instruction with the overflow designator set, the next instruction is skipped; (P) + 2 → P and the overflow designator is cleared. If an overflow condition did not occur on an arithmetic instruction with the overflow designator not set, the next instruction is executed. K is ignored.

5053 TEST NO OVERFLOW (TNO)

Operation: If overflow designator is not set (P) + 2 → P
If overflow designator is set (P) + 1 → P

Execution Time: 1.00 usec.

This instruction tests the condition of the overflow designator and passes program control accordingly. If an overflow condition did not occur on an arithmetic instruction with the overflow designator not set, the next instruction is skipped; (P) + 2 → P. If an overflow condition did occur on an arithmetic instruction with the overflow designator set, the next instruction is executed; (P) + 1 → P, and clears the overflow designator.

5054 TEST ODD PARITY (TOP)

Operation: If sum of ones in [(AU) **AND** (AL)] is odd, (P) + 2 → P
If sum of ones in [(AU) **AND** (AL)] is even, (P) + 1 → P

Execution Time: 2.40 usec minimum (see *NOTE*)

The contents of AU are logically multiplied with the contents of AL and the number of binary 1's in the result is checked for parity. If the number of 1's is odd, the next instruction is skipped; (P) + 2 → P. If the number of 1's is even, the next instruction is executed; (P) + 1 → P. K is ignored.

The contents of AL and AU remain unchanged and in AL and AU.
$(AU)_f = (AU)_i$ and $(AL)_f = (AL)_i$

Example:

(AU) = 000077_8 – Mask

(AL) = 127723_8

[(AU) **AND** (AL)] = 000023_8

Bit Sum = 3

Since the bit sum is odd, the next instruction is skipped.

NOTE:

IOM#0 is used in the execution of this instruction; therefore, the execution time of this instruction is dependent upon queuing within the IOM.

5055 TEST EVEN PARITY (TEP)

Operation: If sum of ones in [(AU) **AND** (AL)] is even, (P) + 2 → P
If sum of ones in [(AU) **AND** (AL)] is odd, (P) + 1 → P

Execution Time: 2.40 usec minimum.

The contents of AU are logically multiplied with the contents of AL and the number of binary 1's in the result is checked for parity. If the number of 1's is even, the next instruction is skipped; (P) + 2 → P. If the number of 1's is odd, the next instruction is executed; (P) + 1 → P. K is ignored.

The contents of AL and AU remain unchanged and in AL and AU.
$(AU)_f = (AU)_i$ and $(AL)_f = (AL)_i$

NOTE:

IOM#0 is used in the execution of this instruction; therefore, the execution time of this instruction is dependent upon queuing within the IOM.

5056 STOP ON KEY SETTING (SK) – Privileged

Operation: Stop if keys designated by K are set.

Execution Time: 1.00 usec.

There are five stop keys on the UNIVAC 418-III maintenance panel and console which, together with this instruction, permit external control of program stops. Bits 4 through 0 of the K portion of this instruction correspond to stop keys 4 through 0 on the maintenance panel and console. For every bit in K_{4-0} that is set to 1, the corresponding stop key is examined. If any of the examined keys are set, the c/a section stops. If K equals 0 or if all the examined keys are not set, the next instruction is executed; $(P) + 1 \rightarrow P$. If K_5 equals 1, the state of K_{4-0} is ignored and processing stops.

Example:

K = 01 (bit 0) stop if stop key 0 is set.
K = 02 (bit 1) stop if stop key 1 is set.
K = 04 (bit 2) stop if stop key 2 is set.
K = 10 (bit 3) stop if stop key 3 is set.
K = 20 (bit 4) stop if stop key 4 is set.
K = 40 (bit 5) stop unconditionally.
K = 03 (bits 1,0) stop if stop key 1 or 0 is set.

NOTES:

- All combinations of octal codes 00 through 77 are valid codes for K.
- This instruction is treated as a no operation while in guard mode.

5060 ROUND A (RND)

Operation: If (A) is positive and $(AL_{17}) = 1$, $(AU) + 1 \rightarrow AL$
If (A) is negative and $(AL_{17}) = 0$, $(AU) - 1 \rightarrow AL$
If otherwise, $(AU) \rightarrow (AL)$

Execution Time: 1.625 usec.

The purpose of this instruction is to round off double-length arithmetic results to single-length. If AL contains a significant bit, the significant bit being 1 for positive numbers and 0 for negative numbers, the magnitude of the AU portion of the double-length result is increased by 1 and the AL portion is discarded. In all cases, whether rounding takes place or not, the contents of AU replace the contents of AL. K is ignored.

$(AU)_f = (AU)_i$

The contents of AU remain unchanged and in AU.
The contents of AL are destroyed.

Example:

$(A) = 120201\ 653375_8$
$(AU) = 120201_8$
$(AL)_i = 653375_8$
$(AL)_f = 120202_8$

NOTE:

If the contents of AU equal positive 377777_8 and the contents of AL_{17} equal 1, or if the contents of AU equal negative 377777_8 and the contents of AL_{17} equal 0 and the ROUND A instruction is executed, overflow occurs and the overflow designator is set. The state of the overflow designator is tested by either the SKIP ON OVERFLOW instruction (f = 5052) or the SKIP ON NO OVERFLOW instruction (f = 5053). The execution of either of these two instructions clears the overflow designator.

5061 COMPLEMENT AL (CPL)

Operation: $(AL) \rightarrow AL$

Execution Time: 1.00 usec.

The contents of AL are complemented and the result is placed in AL. K is ignored.

NOTES:

- This instruction effects a bit-by-bit complement of the contents of AL.
- If the contents of AL are all 0's, the result of the complement is all 0's.

5062 COMPLEMENT AU (CPU)

Operation: (AU) → AU

Execution Time: 1.00 usec.

The contents of AU are complemented and the result is placed in AU. K is ignored.

NOTES:

- This instruction effects a bit-by-bit complement of the contents of AU.
- If the contents of AU are all zeros, the result of the complement is all zeros.

5063 COMPLEMENT A (CPA)

Operation: (A) → A

Execution Time: 1.875 usec.

The contents of A are complemented and the result is placed in A. K is ignored.

NOTES:

- This instruction effects a bit-by-bit complement of the contents of A.
- If the contents of A are all 0's, the result of the complement is all 0's.

5065 LOAD GUARD MODE (LGM) – Privileged

Operation: Load the lower and upper storage registers with $((P) + 1)_{8-0}$ and $((P) + 1)_{17-9}$ and set guard mode designator active; $(P) + 2 \rightarrow P$.

Execution Time: 1.75 usec.

5066 SET AUDIBLE ALARM (SAA) – Privileged

Execution Time: 1.00 usec.

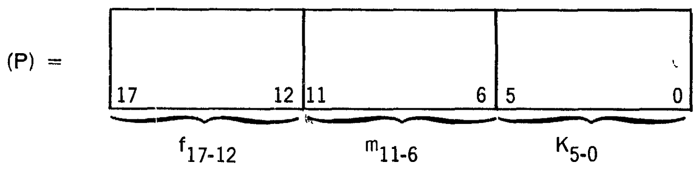
This instruction initiates the console audible alarm. This alarm must be manually reset with the audio reset switch on the console. K is ignored.

5067 ENABLE ESI INTERRUPTS (EEI) – Privileged

Operation: Remove ESI interrupt lockout.

Execution Time: 1.00 usec.

This instruction clears the ESI interrupt lockout designator which is set by the generation of an ESI "hard" interrupt. If the K portion of the instruction is any octal code 00 through 17, IOM#0 is selected; and if the K portion is any octal code 20 through 37, IOM#1 is selected.



NOTES:

- The IOM does not notify the arithmetic section of interrupts tabled while the ESI interrupt lockout designator is set.
- After the execution of this instruction, the next ESI interrupt which is received by the specified IOM generates a "hard" interrupt in addition to being tabled.
- This instruction does not clear the interrupt lockout in the command/arithmetic section but clears only the ESI interrupt lockout in the IOM specified by the K portion of the instruction.
- ESI interrupts are inhibited for one instruction time following the execution of this instruction.

5070 BLOCK TRANSFER (BT)

Operation: If $K \neq 0$, $(AU) \rightarrow (AL)$; $(AU) + 1 \rightarrow AU$; $(AL) + 1 \rightarrow AL$.
The sequence is repeated K times.

Execution Time: $(1.750 + 1.5 \times$ number of words in block) usec.

This instruction transfers the number of words specified by the K portion of the instruction from an initial address specified by the contents of AU to an initial address specified by the contents of AL. The contents of AU equal the source address and the contents of AL equal the destination address. The contents of AU and AL are incremented by 1 with each word transferred.

NOTES:

- The maximum number of words that can be transferred with a single instruction is limited by the K portion of the instruction, 77 octal words.
- If an interrupt is generated during the block transfer, it is not honored until the completion of the BLOCK TRANSFER instruction.
- If K equals 0, no data is transferred, and the contents of AU and AL remain unchanged.
$(AU)_f = (AU)_i$ and $(AL)_f = (AL)_i$

5072 LOAD INDEX REGISTER POINTER (LIR)

Operation: $K_{2-0} \rightarrow$ IRP

Execution Time: 2.50 usec.

The execution of this instruction causes the present contents of the B "hard" register to be stored at the address specified by the present contents of IRP. IRP is then loaded with the low-order three bits, bits 2 through 0, specified by the K portion of the instruction. The contents of storage, specified by the new contents of IRP, are loaded into B.

NOTES:

- The constant K is contained within the instruction and does not refer to an address.
- IRP points to storage address 10_8 when it is loaded with 00_8.
- The index registers, storage addresses 01_8 through 10_8, may be loaded during an initial load operation.

5073 LOAD SPECIAL REGISTER (LSR)

Operation: $y \rightarrow SR_{5-0}$

Execution Time: 1.00 usec.

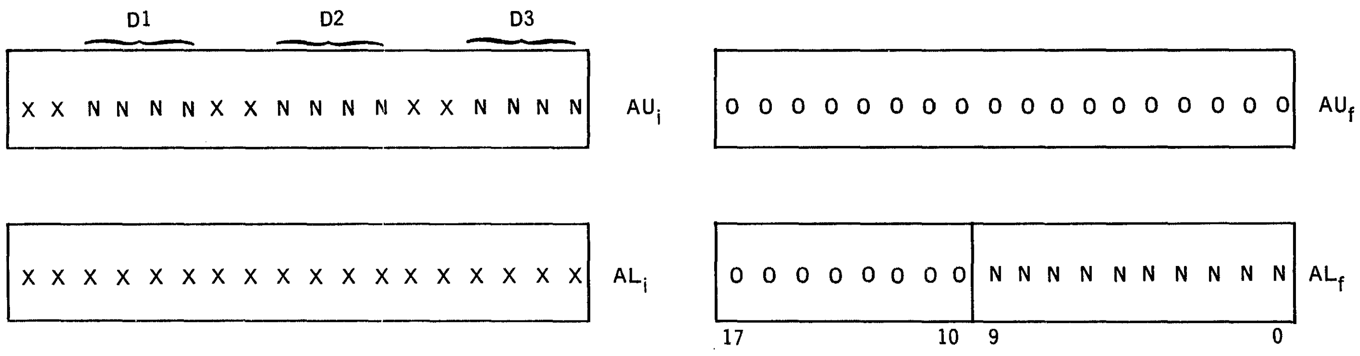
The execution of this instruction causes the low-order six bits of the instruction, specified by K, to replace the contents of the special register. The special register is activated only if bit 4 is set. Bit 5 and bits 3 through 0 define the storage segment to be addressed.

5074 DECIMAL TO BINARY CONVERSION (DB)

Operation: $[(AU_{0-3,6-9,12-15})]_{10} \rightarrow [AL_{9-0}]_2$

Execution Time: 7.735 usec.

This instruction causes a three-character BCD number, packed in a six-bit field in AU, to be converted into a binary number. The resultant binary number is the content of AL. The maximum decimal number to be converted must not exceed $[999]_{10}$.



D1, D2, D3 are assumed to be unbiased, positive BCD digits. XX bits are ignored (D1 = MSC).

NOTES:

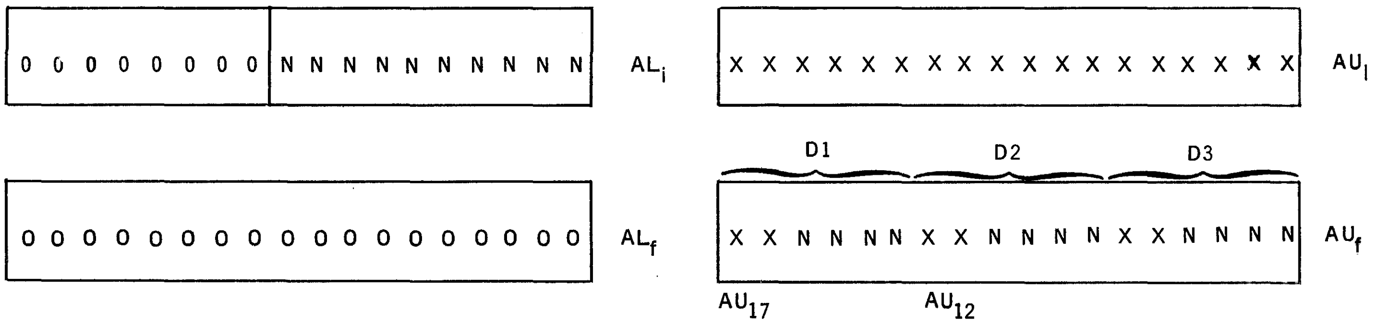
- No test is made for invalid BCD characters; that is, greater than 9.
- This instruction should be useful in program conversion of longer fields by a convert, multiply by 10^N, add, process.
- In processors not equipped with this feature, convert commands are considered a fault and generate a supervisor call interrupt.

5075 BINARY TO DECIMAL CONVERSION (BD)

Operation: $[(AL_{9-0})]_2 \rightarrow [AU_{0-3,6-9,12-15}]_{10}$

Execution Time: 8.250 usec.

This instruction causes a binary number which must not exceed 999_{10} to be converted to BCD. The binary number contained in AL is converted to BCD and is placed in AU in three six-bit characters. The first two bits of each packed character are to be ignored and the next 4 bits contain the BCD code. The most significant character appears at bits AU_{17} through AU_{12}.


0 0 0 0 0 0 0 0 | N N N N N N N N N N
AL_i
X X X X X X X X X X X X X X X X X X
AU_i
D1
D2
D3
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
AL_f
X X N N N N X X N N N N X X N N N N
AU_f
AU_{17}
AU_{12}

NOTES:

- Larger binary numbers should be converted by a divide by 10^3, convert, store sequence.
- In systems not equipped with this feature, the convert commands are considered a fault and generate a supervisor call interrupt.

APPENDIX A. INSTRUCTION REPERTOIRE SUMMARY

OPERATION CODE	MNEMONIC	INSTRUCTION	DESCRIPTION	TIMING IN μ SECONDS
ARITHMETIC COMMANDS				
5060	RND	Round A	If (A) is positive and $(AL_{17})=1$, $AU+1\rightarrow AL$; If (A) is negative and $(AL_{17})=0$, $(AU)-1\rightarrow AL$; otherwise $(AU)\rightarrow AL$.	1.625
14	AL	Add to Lower	$(AL)+(Y)\rightarrow AL$	1.50
15	AL*	Add to Lower		1.50
16	ANL	Add Negatively to Lower	$(AL)-(Y)\rightarrow AL$	1.50
17	ANL*	Add Negatively to Lower		1.50
20	AA	Add to A	$(A)+(Y-1,Y)\rightarrow A$	3.0
21	AA*	Add to A		3.0
22	ANA	Add Negatively to A	$(A)-(Y-1,Y)\rightarrow A$	3.0
23	ANA*	Add Negatively to A		3.0
24	M	Multiply	$(AL)\,.\,(Y)\rightarrow A$	6.5① 7.375②
25	M*	Multiply		6.5① 7.375②
26	D	Divide	$(AL)\div(Y)\rightarrow AL$; Remainder $\rightarrow AU$	6.5③ 7.375④
27	D*	Divide		6.5③ 7.375④
71	ALK	Add to Lower A ''Konstant''	$(AL)+Z\rightarrow AL$	1.0
FLOATING-POINT ARITHMETIC COMMANDS				
5002**	FA	Floating Point Add	$(A)+(Y-1,Y)\rightarrow A$	4.35+x
5003**	FS	Floating Point Subtract	$(A)-(Y-1,Y)\rightarrow A$	4.35+x
5004**	FM	Floating Point Multiply	$(A)\,.\,(Y-1,Y)\rightarrow A$	12.0
5005**	FD	Floating Point Divide	$(A)\div(Y-1,Y)\rightarrow A$	12.0
5006**	FP	Floating Point Pack	Normalize (A),pack with biased characteristic from (Y), and store in A.	3.5+x
5007**	FU	Floating Point Unpack	Unpack A, leave mantissa in A. Store characteristic in Y.	3.5
BINARY/DECIMAL CONVERSION COMMANDS				
5074	DB	Decimal-to-Binary Conversion	$(AU_{15-12,\ 9-6,\ 3-0})\rightarrow AL$ (Binary)	7.375
5075	BD	Binary-to-Decimal Conversion	$AL\rightarrow(AU_{15-12,\ 9-6,\ 3-0})$ (Decimal)	8.250

OPERATION CODE	MNEMONIC	INSTRUCTION	DESCRIPTION	TIMING IN μ SECONDS
LOGICAL COMMANDS				
51	OR	Inclusive OR	(AL) OR $(Y) \to AL$	1.50
52	AND	Logical AND	(AL) AND $(Y) \to AL$	1.50
53	XOR	Exclusive OR	(AL) XOR $(Y) \to AL$	1.50
5061	CPL	Complement A Lower	The complement of $(AL) \to AL$	1.0
5062	CPU	Complement A Upper	The complement of $(AU) \to AU$	1.0
5063	CPA	Complement A	The complement of $(A) \to A$	1.875
TRANSFER COMMANDS				
10	LU	Load A Upper	$(Y) \to AU$	1.50
11	LU*	Load A Upper		1.50
12	LL	Load A Lower	$(Y) \to AL$	1.50
13	LL*	Load A Lower		1.50
44	SL	Store A Lower	$(AL) \to Y$	1.50
45	SL*	Store A Lower		1.50
46	SU	Store A Upper	$(AU) \to Y$	1.50
47	SU*	Store A Upper		1.50
70	LLK	Load A Lower with "Konstant"	$Z \to AL$	1.0
04	MSL	Masked Selective Load	$(Y_N) \to AL$ for $(AU_N)=1$	1.50
05	MSL*	Masked Selective Load		1.50
32	LB	Load Index Register	$(Y) \to IR$	1.5
33	LB*	Load Index Register		1.5
42	SB	Store Index Register	$(IR) \to Y$	1.50
43	SB*	Store Index Register		1.50
36	LBK	Load Index Register with "Konstant"	$Z \to IR$	0.75
37	LBK*	Load Index Register with "Konstant"		0.75
74	SAD	Store Address of A Lower	$(AL_{11-0}) \to Y_{11-0}$	3.0
5072	LIR	Load Index Register Pointer	$K_{2-0} \to IRP$	2.5
5073	LSR	Load Special Register	$K_{5-0} \to SR$	1.0
72	SIR	Store Index Register Pointer	$(IRP) \to Y_{2-0} \begin{Bmatrix} \text{If } (IRP)=0, \\ 001 \to Y_{5-3} \\ \text{If } (IRP) \neq 0, \\ 000 \to Y_{5-3} \end{Bmatrix}$	3.0
75	SSR	Store Special Register and Inactivate	$(SR) \to Y_{5-0}$, $0 \to SR_4$	3.0
40	CY	Clear Y	$0 \to Y$	1.50
41	CY*	Clear Y		1.50
5070	BT	Block Transfer	Transfer K words from $ADR_{AU} \to ADR_{AL}$	1.750+1.5n
5017	SSD	Store Special Designators	$(SD) \to IAR+1$	2.5
5020	LSD	Load Special Designators	$(IAR+1) \to SD$	2.5

OPERATION CODE	MNEMONIC	INSTRUCTION	DESCRIPTION	TIMING IN μ SECONDS
SHIFT COMMANDS				
5041	SRU	Shift Right A Upper	Shift AU right (END-OFF) K bit positions	1 +x
5042	SRL	Shift Right A Lower	Shift AL right (END-OFF) K bit positions	1+x
5043	SRA	Shift Right A	Shift A right (END-OFF) K bit positions	1+x
5044	SCA	Scale A	Shift A left (END AROUND) K places or until normalized K less shift $\rightarrow 00017_8$	2+x
5045	SLU	Shift Left A Upper	Shift AU left (END AROUND) K bit positions	1 +x
5046	SLL	Shift Left A Lower	Shift AL left (END AROUND) K bit positions	
5047	SLA	Shift Left A	Shift A left (END AROUND) K bit positions	1 +x
LOOP CONTROL COMMANDS				
73	JBNZ	Jump and Modify if Index Register Non-Zero	If (IR)$\neq$0, (IR)$-$1$\rightarrow$IR and Y$\rightarrow$IAR If (IR)=0, (IAR)+1$\rightarrow$IAR	1.75
56	TB	Test B-Register for Equality	If(IR)=Y, (IAR)+2$\rightarrow$IAR If(IR)$\neq$Y,(IR)+1$\rightarrow$IR	2.5
57	TZ	Test Any Location for Zero	If(Y)=0,(IAR)+2$\rightarrow$IAR If(Y)$\neq$0,(Y)-1$\rightarrow$Y	2.25
COMPARE COMMANDS				
02	CL	Compare A Lower	(AL): (Y) set CD accordingly	1.50
03	CL*	Compare A Lower		1.50
06	CLM	Compare A Lower Masked by A Upper	[(AU) AND (AL)] : [(AU) AND (Y)]; set CD accordingly	2.0
07	CLM*	Compare A Lower Masked by A Upper		2.0
COMPARISON JUMP COMMANDS (COMPARE DESIGNATOR SET)				
60,61	JE	Jump on Equal	If CD equal condition set, Y$\rightarrow$IAR	0.75
62,63	JNE	Jump on Not Equal	If CD equal condition clear, Y$\rightarrow$IAR	0.75
64,65	JNLS	Jump on Not Less	If CD not less than condition, Y$\rightarrow$IAR	0.75
66,67	JLS	Jump on Less	If CD less than condition, Y$\rightarrow$IAR	0.75

OPERATION CODE	MNEMONIC	INSTRUCTION	DESCRIPTION	TIMING IN μ SECONDS
ARITHMETIC JUMP COMMANDS (COMPARE DESIGNATOR NOT SET)				
60	JUZ	Jump on A Upper Zero	If (AU)=0, Y→IAR	0.75
61	JLZ	Jump on A Lower Zero	If (AL)=0, Y→IAR	0.75
62	JUNZ	Jump on A Upper Non-Zero	If (AU)≠0, Y→IAR	0.75
63	JLNZ	Jump on A Lower Non-Zero	If (AL)≠0, Y→IAR	0.75
64	JUP	Jump on A Upper Positive	If (AU) is positive, Y→IAR	0.75
65	JLP	Jump on A Lower Positive	If (AL) is positive, Y→IAR	0.75
66	JUN	Jump on A Upper Negative	If (AU) is negative, Y→IAR	0.75
67	JLN	Jump on A Lower Negative	If (AL) is negative, Y→IAR	0.75
UNCONDITIONAL JUMP COMMANDS				
34	J	Jump	Y→IAR	0.75
35	J*	Jump		0.75
55	JI	Jump Indirect	(Y_{16-0})→IAR	1.50
30	SLJI	Store Location and Jump Indirect	(IAR)+1→Location in (Y); (Y)+1→IAR	2.25
31	SLJI*	Store Location and Jump Indirect		2.25
76	SLJ	Store Location and Jump	(IAR)+1→Y; Y+1→IAR	2.0
SKIP COMMANDS				
5050	TK	Test Keys	Skip if keys designated by K are set. (IAR)+2→IAR	1.0
5051	TNB	Test No Borrow	If borrow indicator off, (IAR)+2→IAR	1.0
5052	TOF	Test Overflow	If overflow indicator on, (IAR)+2→IAR	1.0
5053	TNO	Test No Overflow	If overflow indicator off, (IAR)+2→IAR	1.0
5054	TOP	Test Odd Parity	If sum of 1's in (AU) **AND** (AL) is ODD, (IAR)+2→IAR	2.4 minimum
5055	TEP	Test Even Parity	If sum of 1's in (AU) **AND** (AL) is EVEN, (IAR)+2→IAR	2.4 minimum
EXECUTIVE COMMANDS (INTERRUPT CONTROL)				
5024 } 5025 }	WFI	Wait for Interrupt	Stop C/A Unit (not I/O) until Interrupt	1.0
5030 } 5031 }	AAI	Allow All Interrupts	Allow all Interrupts	1.0
5034 } 5035 }	PAI	Prevent All Interrupts	Prevent all Interrupts	1.0
54	EJI	Enable Interrupts and Jump Indirect	(Y_{15-0})→IAR; enables Interrupts	1.50
5067	EEI	Enable ESI Interrupt	If K=0 lows ESI Interrupts, IOM #0; If K=20_8, allow ESI Interrupts, IOM #1	1.0

OPERATION CODE	MNEMONIC	INSTRUCTION	DESCRIPTION	TIMING IN μ SECONDS
EXECUTIVE COMMANDS (I/O)				
5011	LIC	Load Input Channel	Load I/O Channel K from (IAR)+1 and (IAR)+2, initiate input; then (IAR)+3 $\rightarrow$ IAR	5.3 minimum
5012	LOC	Load Output Channel	Same as LIC except that output is initiated	5.3 minimum
5013	LFC	Load External Function Channel	Same as LIC except that External Function is initiated	5.6 minimum
5015	STIC	Stop Input on Channel	Stop Input on Channel K	2.15 minimum
5016	STOC	Stop Output on Channel	Stop Output on Channel K	2.15 minimum
5021	TIC	Test Input on Channel	If Input Channel K idle, (IAR)+2 $\rightarrow$ IAR	1.0
5022	TOC	Test Output on Channel	If Output Channel K idle, (IAR)+2 $\rightarrow$ IAR	1.0
5023	TFC	Test External Function on Channel	If External Function Channel K idle, (IAR)+2 $\rightarrow$ IAR	1.0
EXECUTIVE COMMANDS (STORAGE PROTECTION)				
5065	LGM	Load Guard Mode	$((IAR)+1)_{17-9} \rightarrow$ Upper Limit $((IAR)+1)_{8-0} \rightarrow$ Lower Limit, Guard Mode is set and (IAR)+2 $\rightarrow$ IAR	1.75
EXECUTIVE COMMANDS (STOP)				
5056	SK	Stop on Key Settings (if not in Guard Mode)	Stop, if keys designated by K are set; if in Guard Mode, (IAR)+1 $\rightarrow$ IAR	1.0
EXECUTIVE COMMAND (SPECIAL)				
5010**	RS	Read and Set	(Y) $\rightarrow$ AL, 1 $\rightarrow Y_{17}$	2.5
5026	NOP	No Operation		1.0
5066	SAA	Set Audible Alarm		1.0
SUPERVISOR CALL COMMANDS				
00		Supervisor Call		0.75
01		Supervisor Call		0.75
77		Supervisor Call		0.75
5000		Supervisor Call		1.0
5001		Supervisor Call		1.0
5077		Supervisor Call		1.0

LEGEND FOR INSTRUCTION REPERTOIRE

Subscripts specify bit positions in the register or quantity subscripted.
N represents each bit position.

*	= To index an assembler instruction, prefix operand with *(asterisk). The assembler adds 1 to octal operation code, or set $2^{12} = 1$ of IAR + 1.
x	= $\frac{\text{number of shifts}}{8}$ μ seconds
n	= number of words in the block
**	= IR-sensitive if 2^{12} of IAR + 1 is set to 1; indexing is indicated by prefixing the operand with an asterisk (*).
AL	= Lower accumulator
AU	= Upper accumulator
A	= Upper and lower accumulators acting as one register
IR	= The active index register
IAR	= Instruction address register
CD	= Compare designator
Y	= On the left of the → symbol, the storage address in the low-order 12 bits of the instruction (bits 11–0); on the right of the → symbol, the storage location specified by that address.
K	= The unsigned integer or bit configuration in the low-order 6 bits of the instruction (bits 5–0).
Z	= The low-order 12 bits of the instruction, extended to 18 bits by repetition of bit 11 in bit positions 17–12, and treated as a constant in the range –3777 to +3777 octal.
( )	= Contents of the register named in the parentheses; that is, (Y) = contents of Y.
→	= Replaces the contents of
:	= Compare algebraically the quantities on either side of this symbol.
XOR	= Exclusive OR
AND	= Logical AND
OR	= Logical OR

① Multiplying numbers of like signs.

② Multiplying numbers of unlike signs.

③ Dividing positive numbers.

④ Dividing numbers of unlike signs or negative signs.

A sample of each operand type follows. To index an instruction in the assembly language, prefix the operand with an asterisk. The assembler adds a 1 bit to the octal op-code of single word instructions, or sets bit 12 of the second word of two-word instructions (op-codes 5002 through 5010 and 5064).

```
    LABEL          OPERATION              OPERAND                       COMMENTS
1         10         20         30         40         50         60         70
TWO      EQU       2
WORK     +0
TEMP     RES       10
         +0        . FIXED POINT PART
FIX      +0        . FIXED POINT PART
EXP      +0        . EXPONENT
VALUE    RES       20


         LBK       011       . Z OPERAND
         LL        *TEMP     . INDEXED Y OPERAND
         ALK       TWO       . Z OPERAND
         SL        WORK      . Y OPERAND
         SLL       1         . K OPERAND
         SL        $+2       . Y OPERAND
         J         OUT       . Y OPERAND


                   . FOR EACH OF THE FLOATING POINT INSTRUCTIONS
                   . THE ASSEMBLER GENERATES TWO LINES OF CODE
                   .
                   LU        FIX-1
                   LL        FIX
                   FP        EXP
                   FA        *VALUE    . CONTENTS OF VALUE MUST BE
                                       . IN FLOAT FORMAT
```

Library 3   3   1004
Univac Marketing Education Center
Div Sperry Rand Corp   52;56;58;61
Valley View Dunhill Bldg
251 West DeKalb Pike
King of Prussia Pa 19406

UNIVAC

418-III

ASSEMBLER
UP-7599 REV. 1

UNIVAC SYSTEMS PROGRAMMING LIBRARY SERVICES **REVISION**

UNIVAC 418-III Real-Time System Library Memo 18 announces the release and availability of "UNIVAC 418-III Real-Time System RTOS Assembler Programmers Reference," UP-7599 Rev. 1, covers and 171 pages. This is a Standard Library Item (SLI).

This version of the UNIVAC 418-III Assembler manual describes the language and its uses in more detail than the original. Included are descriptions of the coding format, expressions, directives, PROC's, and paraforms.

Certain directives have been removed and some new more powerful ones have been added. For example the UNLIST directive has been added to the assembler, it provides a means of selectively preventing the printing of output of sections of a program.

Dimensioned Labels, a new feature, are also described in detail. These are labels which are distinguished by their subscripts rather than by the label itself.

Sample assembled printouts are included wherever possible to support explanations and show examples of the features discussed. Coding examples are also given throughout the manual to assist in a logical presentation and flow.

A detailed explanation of the instruction repertoire is included as well as an instruction repertoire summary.

Destruction Notice: UP-7599 Rev. 1 supersedes and replaces "UNIVAC 418-III Real-Time System Assembler Programmers Reference," UP-7599, released on Library Memo 1 dated June 24, 1968. Please destroy all copies of UP-7599 and/or Library Memo 1.

Distribution of UP-7599 Rev. 1 has been made as indicated below. Additional copies may be requisitioned from Holyoke, Massachusetts via a Sales Help Requisition through your local Univac Manager.

Robert O. Wallace
GROUP MANAGER
Documentation and Library Services

NOTE: Back Orders for this item are being filled automatically. Please do not reorder.

TO LISTS: 10U, 217, 630, and 692 Library Memo 18 only.	ATTACHMENTS: UP-7599 Rev. 1 plus Library Memo 18 to S.P.L.S. Lists 57 and 58.	THIS SHEET IS: UNIVAC 418-III Real-Time System Library Memo 18.
		DATE: July 23, 1970

UNIVAC

SPERRY RAND

UP-7599 Rev. 1