A GUIDE TO ASSEMBLY LANGUAGE PROGRAMMING FOR THE UNIVAC 1108

by

R. J. Ciecka
and
G. R. Ryan

UNIVERSITY OF MARYLAND
COMPUTER SCIENCE CENTER
COLLEGE PARK, MARYLAND

COMPUTER NOTE CN-2 SEPTEMBER 1971

A GUIDE TO ASSEMBLY LANGUAGE PROGRAMMING FOR THE UNIVAC 1108

by

R. J. Ciecka

and

G. R. Ryan

OFFICE OF USER SERVICES
COMPUTER SCIENCE CENTER
UNIVERSITY OF MARYLAND
COLLEGE PARK, MARYLAND 20742

INTRODUCTION

This manual, originally prepared by R. J. Ciecka and G. R. Ryan and presented through the Department of Electrical Engineering of the University of Maryland on March 1, 1971, is herewith issued by the Computer Science Center of the University of Maryland under the reference number CN-2. The departmental identification assigned by Electrical Engineering is I0010D, which replaces that department's manual I0010C.

References used in the preparation of this guide include: UP-4053; UP-4040; UP-4042; and the University of Maryland User Reference 70-01.

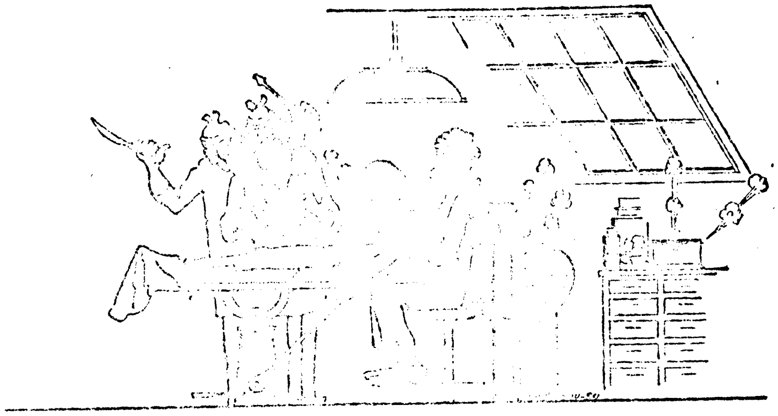
The following is the original introduction to the manual:

This manual provides a concise and relatively complete guide to the UNIVAC 1108 Assembler. It is designed primarily for the student who is having his first contact with 1108 assembly language, but will also serve as a handy reference for the more advanced programmer. Information from as many sources as could be found had been combined and condensed so that for the first time (to our knowledge) the user can find information on assembly language subroutine linkage, input/output, and diagnostic processors presented in a clear manner. Those users who find that they need still more detailed information should consult the UNIVAC and U of M references listed at the beginning of this manual.

It is a pleasure to acknowledge the many and varied contributions of the University of Maryland's Computer Science Center Systems Staff. In particular we would like to thank Ray Cook of the Systems Staff and Professor Marshall D. Abrams of the Department of Electrical Engineering.

We hope this manual meets the needs of those who use it. Good luck.

TABLE OF CONTENTS



[illegible] MANUAL.

1. Calling the Assembler

Under EXEC 8 the format of the assembly control card is:

@ASM, <options> <field 1>, <field 2>, <field 3>

The available options are coded as follows:

C Produce symbolic listing (no octal).
D Produce double-spaced listing.
M Request 10K additional core for symbol and procedure sample table.
N Suppress all listing.
O Produce octal listing only (no symbolic).
S Produce octal and symbolic listing (Normal listing option).
T Request 5K additional core for symbol and procedure sample table.
U Update and produce new cycle of source element.
I Insert new element to program file from control system.
W List corrections.

<field 1> is the input source file and element.
<field 2> is the relocatable file and element.
<field 3> is the updated source file and element.

If the I option is on (as when inserting from cards), specification <field 1> names the program file to contain the source code. If assembling from tape, <field 1> is the file name of that tape, <field 2> is the relocatable program element name, and <field 3> specifies the name of the program file to contain the source code. If file names are not specified, the temporary run file is utilized. If assembling from tape and the tape is positioned incorrectly (to an element other than the one specified) an error is produced.

2. Basic Assembler Language

2.1 Computer Instruction-Word Format

Every machine instruction for the 1108 adheres to the following format:

f	j	a	x	h	i	u
35 30	29 26	25 22	21 18	17	16	15 00

Where:

- f specifies the function code.
- j specifies the partial word designator or minor function code, if any.
- a specifies the control register or input/output channel, if any.
- x specifies the index register, if any.
- h specifies index modification and if set calls for address modification.
- i specifies indirect addressing.
- u specifies the address field.

2.2 Assembler Format

In writing instructions using the 1108 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 segment.

All of the fields and subfields following the label field in the 1108 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 his symbolic coding. It is highly recommended that within the confines of a given program, the programmer keep a fixed set of column conventions for the sake of legibility.

The basic line of coding is divided into 3 or fewer fields, called label, operation, and operand fields. A field is terminated by one or more spaces and may be divided into subfields. A subfield is an expression which is terminated by a comma followed by zero or more spaces. The last subfield in the field, of course, is terminated by the space (at least one) that terminates the field.

The format of a symbolic instruction differs from the computer instruction word for convenience of programming as follows. Commas separate subfields.

LABEL FIELD	OP FIELD	OPERAND FIELD
	F F, J	A, U, X, J A, U, X

In addition to instructions of the type discussed above, there are several which do not use the A field. The operands of such instructions comprise the U, X, and J subfields.

LABEL	OP	OPERAND
	F F, J	U, X, J U, X

2.3 Description of Fields

2.3.1 Label Field

The label field, where used, must start in column one and terminate with a blank. It may contain a declaration of a specific location counter or a label or both, as explained below.

2.3.1.1 Labels

A label is a means of identifying a value or a line of symbolic coding. It consists of an alphabetic character which may be followed by as many as eleven alphanumeric characters (A through Z and 0 through 9). When a label is used, it must begin in column one and terminates with a blank.

In addition to the alphanumeric characters, the $ may be used in a label beginning with the second character. However, the use of the $ is limited because references to the Executive System are made via system's labels which utilize the $ in various character positions (see "1108 Executive System, Programmer's Reference Manual", UP-4144).

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

2.3.1.2 Location Counter Declaration

There are 32 location counters in the 1108 Assembler, any one of which may be used or referenced in any sequence. These counters provide information required by the collector to regroup lines of coding in any specified manner. This regrouping capability enables isolation of constants or instructions, or components of each which in turn gives great flexibility to segmentation. A specific location counter is declared by writing $(e) as the first entry in the label field, e being the location counter number (0 through 31). Any change to an unnamed location counter affects the counter currently in control. A specified location counter remains in use until a new location counter is declared. If no location counter is explicitly specified, the program is controlled by location counter zero. Any time a location counter is specified, all subsequent coding falls under its control. To include a label on the same line as a change-of-location-counter item, one must place a comma between the closing paren and the label, with no imbedded blanks (e.g., $(2), LABEL).

Each new location counter entry begins the coding relative to zero. Coding resumed under a counter that has been used previously continues at the last address specified for that counter.

2.3.1.3 Location Counter Reference

Reflexive addressing may be achieved by referencing the current location counter, or a specific location counter, within a symbolic line. The symbol for a current location counter reference is $. When the assembler encounters $ it inserts the value of the controlling location counter. A reference to a specific location counter is made by $(e), where e denotes the specified location counter. In this case the assembler substitutes the value of location counter e for the symbolic reference. When $+b is coded care should be taken so that the source-coded interval b does not extend over a procedure call. This is particularly a problem if the procedure called may generate a variable number of lines of code.

It is standard programming practice to assemble the instructions under odd location counters and the data under even location counters.

2.3.2 Operand Field

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

Most operands may contain fewer than the maximum number of subfields implied by the operation field. If a subfield other than the normal first or last is to be omitted, two continguous commas should be used to denote that subfield (e.g., ,,). If the last subfield or subfields are to be omitted, no comma may appear immediately following the last coded subfield. A period space coded just after this subfield stops scanning and speeds up assembly time (e.g., .5).

2.3.2.1 Function Code Designator, f

The machine language function code, or f designator, contained in the leftmost six bit positions, specifies the particular operation that is to be performed. In instructions where $f > 70_8$, the j designator becomes part of the function code.

2.3.2.2 Operand Qualifier or Minor Function Code Designator, j

When $f < 70_8$, the j designator determines whether an entire operand, or only a part of it is to be transferred to or from the arithmetic section. As previously mentioned, in instructions where $f > 70$, j serves as a minor function code rather than as an operand qualifier. When $f = 70$, the j-designator combines with f to form the function code, and may not be coded.

As an operand qualifier in the case of partial word transfers to the arithmetic section, j specifies which half-word, third-word, or sixth-word is to be utilized. The transfer is always to the low order positions of the arithmetic section. In transfers from the arithmetic section, j specifies into which half-word, third-word or sixth-word the low order positions of the word in the arithmetic section will be transferred.

In half-word transfers to the arithmetic section, j can specify whether sign extension is to take place. If it is specified by coding j as 3 or 4, the most significant bit of the half-word fills positions 35 thru 18 of the control register. If sign extension is not specified, i.e., j is coded as 1 or 2, positions 35 through 18 are zero filled.

Sign extension always occurs for third-words and never occurs for sixth-words. No sign extension occurs for transfers from the control registers.

The mnemonic letter codes used in assembly language corresponding to the numerical j designators are given below.

When j equals 16 or 17, the u-field of the instruction becomes the effective operand rather than the address of the operand. When j is coded as 17, sign extension is effective.

J-designators are totally ignored when "U" is a control register, except for 016 & 017, which behave normally.

2.3.2.3 The Register Designator Field, a

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

2.3.2.3a A-Register Designator, A(a)

The a-designator normally specifies a control register location. For arithmetic operations and some other operations which do not specifically reference other registers, the a-designator specifies one of the 16 A-Registers.

2.3.2.3b X-Register Designator, X(a)

The a-designator is also used to reference any one of 15 index registers in control memory. An X-Register is implied by the function code in certain instructions. Control register 000000 cannot be normally referenced by an a-designator.

2.3.2.3c R-Register Designator, R(a)

The a-designator is used to reference any one of 16 R-registers. An R-register is implied by the function code in certain instructions.

Note: Any time a repeat count instruction is executed (such as BT, and all search instructions) the repeat count must be in R1. Univac documentation does not mention this!

2.3.2.4 Definition of Registers In Assembler Programs

A procedure in the library is available which when called by

AXR$

will define symbols for the useable user register set as follows:

Ai, i=0, 1, ..., 15	are defined for accumulators.
Xi, i=1, 2, ..., 11	are defined for index registers.
Ri, i=1, 2, ..., 15	are defined for R-registers.

The accumulators A0 through A3 may also be used as index registers, corresponding to X12, X13, X14, X15 respectively. Also the j subfield of an instruction is defined by the AXR$ procedure as follows:

H1 and H2 refers to

H1	H2

XH1 and XH2 refers to

H1	H2

, sign extension

T1, T2, T3 refers to

T1	T2	T3

Q1, Q2, Q3, Q4 refers to

Q1	Q2	Q3	Q4

(Note: quarter-word references may only be used in special circumstances)

S1, S2, S3, S4, S5, S6 refers to

S1	S2	S3	S4	S5	S6

U refers to immediate operand

XU refers to immediate operand, sign extension

W refers to whole word operand

2.3.2.5 Index Register Designator, x

The format of the indexing information stored at the control register address specified by the x-designator is shown below. Bits 17-00 (X_M) contain the address modifier which is added to the u address; bits 35-18 (X_I) contain an increment which may, if desired, be used to change the value of X_M. This increment may be positive or negative.

35 X_I 18	17 X_M 00

2.3.2.5a Index Register Incrementation Designator, h

When the h-designator is coded as 1, the value of X_M in index register X is increased by the value of X_I. This incrementation takes place during the instruction; after the addition of u and the index register, in forming the effective address. When h is 0, no incrementation takes place.

The entry in the X subfield represents the specific index register to be used. Index register incrementation is indicated in assembly language by means of an asterisk preceding the X subfield (e.g. *X). The 1108 is a one's complement machine and does pre-indexing. This means it increments first and then performs the rest of the instruction.

2.3.2.6 Operand Address or Operand Designator, u

For all instructions the u field specifies an operand for the particular instruction involved. For every instruction cycle the "effective u" must first be calculated. If no address modification, then the coded u field is the effective u. If address modification is specified (by an entry in the X field) then the right half of the specified index register is added to the coded "u" and the result becomes the effective u. For the case of indirection, see the section below.

2.3.2.6a Indirect Addressing Designator, i

The i-designator specifies either direct or indirect addressing of the operand. If i is coded as 0, direct addressing is specified, and u is the effective address of the operand. If i is coded as 1, indirect addressing is specified. Bits 21-00 of the u-addressed operand replace bits 21-00 in the current instruction. Since the 22 bits include the x, h, i, and u-designators, all indexing, index register incrementation, and indirect addressing operations can be cascaded until the i-designator in one of the temporarily formed instructions is 0. If $j<16$, normal partial-word operations on the contents of the address specified by u are performed at the end of cascading. If $j=16$ or 17, cascading is halted when either the i-designator or the x-designator, or both, become zero; the value in u 17-00 becomes the actual operand. Thus, for $j=16$ or 17, indirect addressing is not only conditioned by the i-designator, but is also conditioned upon the x-designator being a non-zero value.

The entry in the U subfield represents the operand base address. Indirect addressing is indicated by means of an asterisk preceding the U subfield (e.g. *U).

2.4 Continuation

If a semicolon (;) is encountered outside of an alphabetic item, the current line is continued with the first non-blank character on the following line. Any characters on the line after the ; are not considered pertinent to the program assembly, and are transferred to the output listing as comments. A semicolon should not be used within a comment unless it is desired to continue that comment on the next line. If a line is broken within a subfield, the semicolon must immediately follow the last character of the previous line, with no intervening blanks.

2.5 Termination

A period followed by a blank (.) terminates a line of coding except when it occurs inside an alphabetic item. Any additional subfields implied by the operation field are taken to be zero. The space following the period avoids confusion with the notation for floating point numbers which use the period without a space. A continuation or termination mark may occur anywhere on a line except as noted above. Following the information portion of a line, any characters may be entered as comments except the apostrophe (').

2.6 Ejection of Paper

A slash (/) appearing in column one advances paper in the printer to the top of the next page. This same line may also contain a line of coding with the label field starting in column two. If it is desired to use the remainder of the line as a comment, a period must follow the slash.

2.7 Data Word Generation

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

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

If the operand field contains one subfield immediately followed by a D or a value greater than 36 bits in length, the 1108 assembler generates a seventy-two bit value contained in two consecutive thirty-six bit computer words. The seventy-two bit value is signed and right-justified.

2.7.1 Expressions

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

2.7.1.1 Elementary Items

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

2.7.1.2 Octal Values

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

For example,

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

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

2.7.1.3 Decimal Values

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

For example,

+ 12	PRODUCES OCTAL WORD 000000000014
+2048	PRODUCES OCTAL WORD 000000004000
-04162	PRODUCES OCTAL WORD 777777767675

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

2.7.1.4 Alphabetic Items

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

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

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

2.7.1.5 Floating Point and Double Precision

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

+16384.0	PRODUCES	FLOATING-POINT WORD 217400000000
±16384.0D	PRODUCES	201740000000 000000000000
19.0D	PRODUCES	200546000000 000000000000

3. Subroutine Linkage

The following information pertains to the FØRTRAN defined standard subroutine linkage. By following the FØRTRAN conventions, an assembly language program may link to, and be linked to, a program unit written in another language. It should be noted that X11 must be used for all subroutine and function linkages with system defined subroutines and functions. Thus, it may be necessary to save the contents of X11.

3.1 Calling Sequence

A subroutine, SUB, with i arguments would be called from FØRTRAN by the statement

```
CALL  SUB(<ARG1>,<ARG2>,...,<ARGi>)
```

or, if SUB was a function-type subprogram, by

```
<variable> = SUB(<ARG1>,<ARG2>,...,<ARGi>)   .
```

The corresponding assembly language code, expressed in Bacus Normal Form, is

```
        LMJ   X11,SUB
         +    <ARG1>
         +    <ARG2>
         .
         .
         .
         +    <ARGi>
         +    <line identification>,<walk-back packet>
```

The names used in the call have the meanings described below.

<ARGi> is the symbolic label assigned to the i^{th} argument
<line identification> is the number assigned to the subroutine call for identification purposes. The assembly language programmer may use any (small) integer.
<walk-back-packet> is the symbolic label assigned to a two word sequence, described below, which EXEC 8 uses in case of an error.

The assembly language program must contain a <walk-back-word>, as the last word in the subroutine linkage is called. Upon return from the subroutine, execution will begin with the word immediately after the walk-back word. Note that for a subroutine with i arguments there are i+1 words after the LMJ. If the subroutine called wanted to load <ARG2> into A0 the form of the assembly code would be

```
        LA    A0,*1,X11
```

3.1.1 Abnormal Return

If an argument is to be specified as an abnormal return (in FØRTRAN, $<statement label>) then the corresponding word in the assembly language subroutine linkage would be

```
        J     <label>
```

where <label> is the symbolic label to which control is to pass if an abnormal return is made.

3.1.2 Function Value Return

If the subprogram is function-type the calling program expects to find a result in A0. (If the subprogram is double precision, the result is in A0 and A1.) The subprogram must leave the calculated result in A0 before returning. It is the job of the calling program to retrieve the result left in A0.

3.1.3 Normal Return

If an argument is to be specified as a normal return, then the corresponding word in the assembly language subroutine linkage would be

```
        J     i+2,X11
```

where i is the number of arguments.

3.2 Use of Registers by Subroutines

A subprogram may use accumulators A0 through A5 and R-registers R1 through R3 without saving them. All other registers used in the subprogram must be saved upon entry and be restored before return.

3.3 The Walk-Back-Packet

The walk-back-packet is a two word pair of locations which are referenced by every subroutine call. These words contain information and are not executable. The first word is the Fieldata name of the program unit. The second word is zero if the program unit is a main program. If the program unit is a subprogram, the second word should contain the contents of X11 upon entry to the subprogram.

For example, in a subroutine RTNE the following assembly language sequence might be used

```
WBCK$    'RTNE'
         +  0
RTNE*    SX      X11,WBCK$+1
```

4. Termination of Execution

It is bad form to terminate execution by "running off the end of the program." Two ways to return control to EXEC 8 are:

ER EXIT$	- If no errors, a normal exit occurs.
ER ERR$	- The A, X, and R registers will always be dumped upon exit.

In case you are wondering, ER stands for "Executive Request."

5. Assembler Directives

The symbolic assembler directives within the 1108 Assembler control or direct the assembly processor just as operation codes control or direct the central computer processor. These directives are represented by mnemonics which are written in the operation field of a symbolic line of code. The general format for directives is,

<label> DIRECTIVE <value>

though all directives do not necessarily include all three fields. Of the fifteen directives available, only a few are discussed here.

5.1 The Reserve Directive, RES

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

<label> RES <value>

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

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

5.2 The END Directive, END

The processing of an END directive indicates to the 1108 Assembler that it has reached the end of a logical sequence of coding. The format is:

```
END        <starting Label>
```

An END line must not include a label.

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

5.3 The Equate Directive, EQU

The EQU directive equates a label appearing in its label field to the value of the expression in the operand field. It is possible to generate a double precision equate statement by having the operand contain one numeric subfield immediately followed by the letter D. The EQU must include all three fields.

```
LABEL     EQU     VALUE
```

A value so defined may be referenced in any succeeding line by the use of the label equated to it. If a label is to be assigned a value by the programmer, it must appear in an EQU line before it is used or referenced in subsequent lines of symbolic coding. Otherwise the label is considered undefined.

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

5.4 LIST and UNLIST Directives

The LIST and UNLIST directives allow the programmer to control the listing of the assembler. The LIST directive allows the programmer to override the effect of; no options on the ASM control card, the N option on the ASM control card, or a previous UNLIST directive that suppressed the listing. Likewise the UNLIST directive allows the programmer to override the effect of the S option on the ASM control card or a previous LIST directive. It should be noted that:

1. LIST and UNLIST directives may be used in the program as often as desired, but must be removed in order to obtain a complete program listing.
2. The UNLIST directive image is not printed.
3. No label or operand is used.

The format is:

```
LIST
UNLIST
```

5.5 The FORM Directive, FORM

The FORM directive is used to set up a special word format which may include fields of variable length. The format is:

LABEL   FORM $<F_1>, <F_2>, \ldots, <F_i>, <F_n>$

The operands $<F_i>$ specify the number of bits desired in each field. The sum of the n values of F_i must equal 36 or 72 depending on whether a single or a double precision form word is desired.

By writing the label of the FORM directive, the form defined in that line of coding may be referenced from another part of the program. The label of the FORM line is written in the operation field and is followed by a series of expressions in the operand field. The expressions in the operand field specify the value to be inserted in each field of the generated word or words. When referencing the FORM directive an E flag will be set if either n F_i's are not supplied or if the number assigned to a particular F_i is larger than the number of bits specified in the FORM statement.

6. Input/Output

Input/output is most easily accomplished via the FØRTRAN formatted input/output package. The following discussion will assume that you are familiar with input/output from FØRTRAN or MAD.

6.1 The Format Specification

Unlike FØRTRAN, where there exists a special statement to create a format specification, assembly language creates a format specification by enclosing it in primes. The format includes the opening and closing parenthesis. The format is referenced by the symbolic name located in the label field on the line of code. The form is

<label>   '(<format specification>)'

For example:

```
FRMT      '(7H0SAMPLE, E10.4, 3I9)'
```

6.2 Assembler Output

6.2.1 Line Printer Output

When output is to occur on the printer, the following three words are used to call the appropriate output subroutine

```
LMJ       X11, NPRT$
 +        1, <format label>
 +        <walk-back word>
```

The executive request PRINT$ may also be used for line printer output. An example of the coding for PRINT$ using a FORM directive is:

```
P       FORM     12,6,18 ---- this sets up
IMAGE   'HO HUM'
LA      A0,(P  5,1,IMAGE) ---- this sets up
ER      PRINT$
```

35 24	23 18	17
# lines to skip	# words to print	from where

This will cause the printer to skip 5 lines and print HO HUM on a single line.

If you want to be tricky you can forget the FORM directive and do the following:

```
IMAGE   'HO HUM'
LA      A0,(0501,IMAGE)
ER      PRINT$
```

This will do the same thing as the previous example.

6.2.2 Output To Other Devices

When output is to go to a legal unit other than the line printer the assembly language code sequence is:

```
LMJ     X11,NWDU$
 +      1,(<unit>)
 +      0,<format label>
 +      <walk-back word>
```

6.3 Assembler Input

6.3.1 Reader Input

For input from the card reader, teletype, or run-stream, the sequence is

```
LMJ     X11,NRDC$
 +      1,<format label>
 +      <walk-back word>
```

An alternate way of reading is by the executive request READ$. The assembly language code would be:

```
LA      A0,(<transfer label>,<starting address label>)
ER      READ$
<starting address label>   RES   <value>
```

where: <transfer> label is where to go when an end-of-file is read
<starting address label> is the base address of the storage area
<value> is the size of the storage area

At the completion of the executive request H2 of A0 will contain the number of words read.

6.3.2 Input With END Clause

Corresponding to the FØRTRAN input

READ (<unit>, <format label>, END=<transfer label>)...

the assembly language code sequence is

```
LMJ     X11,NRDU$
 +      2,(<unit>)
 +      0,<format label>
 +      <walk-back word>
 +      2,<transfer label>
```

6.4 Performing the Input/Output

Once the format has been transmitted, the variable location for each variable is transmitted by the pair

```
LA,U    A0,<variable reference>
SLJ     NIØ1$
```

When all the variables have transmitted, the following line executes the output

```
SLJ     NIØ2$
```

7. Simple Assembly Procedures

7.1 The Functioning of Procedures

There are times while programming in assembly language when it becomes necessary to repeat blocks of code which are virtually identical except for several common subfields of the instruction, e.g.:

```
I)  TLE,U   A0,'9'+1          II)  TLE,U   A1,'9'+1
    TG,U    A0,'0'                 TG,U    A1,'0'
    J       NØTNUM                 J       ALPHA2
```

In both cases, the net effect is to test a given register to see if it has a field-data number in it, and if not, transfer to some location. Now, if the program required many repetitions of this code in many different places, then just the task of writing it would be burdensome, and moreover, if others were to look at such a program, then its sheer bulk might very well be detrimental to their understanding the program flow. Thus it would be very helpful indeed if there were some way we could specify the skeleton of a block of code (a template, so to speak), and then reference that code by a short statement.

7.2 Creating a Procedure

The 1108 Assembler has the capability of being given a block of skeleton statements which may be referenced, and thereby be inserted into the object code, by a single statement. This is effected by the use of the assembler directive PRØC (short for Procedure, the Univac equivalent of what the rest of the world calls a macro). We first give an example of a simple PRØC, and then the form and use of PRØCs in general.

If, using the above example, we placed at the beginning of the program the following skeleton:

```
P*      PRØC
        TLE,U   P(1,1),'9'+1
        TG,U    P(1,1),'0'
        J       P(1,2)
        END
```

then the single statement

```
P       A0,NØTNUM
```

would generate a block of code equivalent to I) above, and

```
P       A1,ALPHA2
```

would produce block II).

The general form of a reference to a PRØC skeleton is:

<PRØC-NAME> <ARGLIST>

where

<ARGLIST> has the form

<field1> <field2> ... <fieldk>

and where the i^{th} field has the form

<subfield1>, <subfield2>, ..., <subfieldM>

where

<PRØC-NAME>

is a name associated with a given skeleton and the fields and subfields are 'arguments' with which the assembler will fill out the skeleton. It is important to note that fields are separated by blanks and subfields of a given field by commas.

In the above example, the PRØC name is P, and there is only one field which has two subfields. Another example would be:

```
JUMP    A5    LA0100,LA0200,LA0300,LA0400    ERR350
```

In this case, JUMP will be a name for a PRØC skeleton, and there are three fields, the first of which has only one subfield, the second four, and the third one.

7.3 Declaring a Procedure

Now that the syntax of a skeleton has been established, it would be beneficial to know how to tell the assembler that something is indeed a PRØC skeleton. This is done by evoking the assembler directive PRØC. The form of declaring a PRØC skeleton is

```
<PRØC-NAME>*            PRØC
```

where <PRØC-NAME> is the name to be attached to the skeleton and starts in column one. As soon as the assembler encounters a PRØC card, all cards thereafter are considered part of the skeleton until the PRØC's associated END card is encountered. This END card is included in addition to the program END card and is needed for every PRØC to signify the end of a logical block skeleton.

7.4 Using a Procedure

All we need now is the mechanism for picking up the arguments to be inserted into the skeleton. This is done by using what Univac calls paraforms (which is indeed quite surprising, as that is the correct term). A paraform has (forgive me) the form

```
<PRØC-NAME>(i, j)
```

where i and j are integers greater than zero. This paraform references the j^{th} subfield of the i^{th} field on the line which referenced the PRØC. Using our first PRØC example:

```
P*      PRØC
        TLE, U    P(1, 1), '9'+1
        TG, U     P(1, 1), '0'
        J         P(1, 2)
        END
```

and the call

```
P       A0, NØTNUM
```

we have that: the paraform P(1, 1) references the first subfield of the first field of the call, i.e., A0, and the paraform P(1, 2) is equated to the value of the second subfield of the first (and only!) field of the call, namely, NØTNUM. In our second sample call, the various arguments would be referenced by:

```
    A5   =   JUMP(1, 1)
LA0100   =   JUMP(2, 1)
LA0200   =   JUMP(2, 2)
LA0300   =   JUMP(2, 3)
LA0400   =   JUMP(2, 4)
ERR350   =   JUMP(3, 1)
```

8. The Assembler Code Listing

Accompanying the symbolic listing of your assembly language program is an octal listing of the code generated by the assembler in instruction format. The reason for this special format is that it is easier to see what has been outputed if each field is separated instead of compressed into the twelve octal bits as in a dump. This format comes out exactly as it appears in an instruction word - i. e., f, j, a, x, h, i, u.

For example:

Assuming LOC is program relative 043, we have

```
27   00   13   00  0  0  000043       LX   X11, LOC
```

where

27 is the function code for LX (as can be found in Appendix B).
13 is the register to be loaded (X11 - Remember, it is octal).
and 43 is LOC.

Remarks:

1) As the A field of an instruction word is only four bits, those instructions requiring A and R registers cannot have the actual address loaded in so their designations are used instead - e.g., 3 for A3 instead of 017. What actually happens is that the assembler subtracts from the actual address 014 for A registers and 0100 for the R's.

Thus

```
10   00   05   00   0   000043       LA   A5, LOC
```

since A5 is at location 021 which would not fit in four bits.

2) The h, i field digit will assume only one of the values 0, 1, 2, 3 as it represents a two bit field.
So,

0 => neither h nor i bits set
1 => only i bit is set
2 => only h bit is set
3 => both h and i bits are set

e.g.,

```
01   00   03   02   0   000043     SA   A3, LOC, X2
01   00   03   02   1   000043     SA   A3, *LOC, X2
01   00   03   02   2   000043     SA   A3, LOC, *X2
01   00   03   02   3   000043     SA   A3, *LOC, *X2
```

3) In the case where immediate addressing is specified, i.e., the J-designator=016 or 017, the h, i field digit is no longer given,

but the six u-field digits represent the entire right half of the word instead of the usual right most 16 bits, thus:

```
10  16  01  00  777776   LA,U   A1,-1
```

instead of

```
10  16  01  00  3  177776   LA,U   A1,-1
```

as older versions of the assembler produced.

For further discussion of this topic see subsection 2.3.2.1 to 2.3.2.6a.

9. Diagnostic Processors

There are on the 1108 two prime vehicles for obtaining dumps of one's program area, the post-mortem-dump (PMD), and the dynamic (snapshot) dump. By far, the most commonly used of the two is the PMD, but the PMD allows one to see one's program area only after execution, and if, as is often the case, the dump is being used for diagnostic purposes, the state of the PMD will only show the program area after the damage has been done and will often not reflect at all the initial course of the problem. Because of this, one may make use of the dynamic dump capabilities. Dynamic dumps allow the assembly language programmer to, at will, selectively dump registers, programs, and/or data areas during execution.

9.1 Obtaining a Snapshot Dump Via X$DUMP

The code for generating the snapshots is inserted into the object code by referencing the system procedure X$DUMP in the source program. The format of the procedure reference is:

```
X$DUMP   <ADDR>,<LENGTH>,'<FORMAT>','<REG.LIST>'
```

where:

<ADDR> is the first address of the area to be dumped.
<LENGTH> is the number of words to be dumped.
<FORMAT> specifies the format of the dump (registers however are always dumped in octal).
<REG. LIST> is any combinations of the letters A, X, or R, specifying A registers, X registers, and R registers respectively.

e.g.,

```
X$DUMP   GARK,10,'S','XR'
```

will dump 10 (decimal) words in instruction format starting at GARK and will be preceeded by an octal dump of all X and R registers.

Only the first two subfields need be coded, in which case, no register will be given and the snapshot will be given in octal (the default format option).

These are seven system defined formats available for use:

'S'	-	(4S30) - instruction format
'Ø'	-	(8Ø14) - octal
'A'	-	(16A6) - alphanumeric
'I'	-	(8I14) - integer
'F'	-	(8F14.8) - fixed decimal
'E'	-	(8E14.8) - floating decimal
'D'	-	(4D28.18) - double floating decimal

Hints for using snapshot dumps:

1) Care should be taken when using instructions such as J $+5, JGD A8, $-15 around X$DUMP procedure calls, as the assembler generates four words of object for each X$DUMP reference.

2) Usually, the most helpful (and most often, the only helpful) use of snapshots is to dump sets of registers at selected points in the program, in which case one would use a reference such as:

```
X$DUMP    BURF,1,'Ø','A'
```

The first two subfields are necessary, since no dump would be taken if the count were zero or not coded. One also usually codes a 1 for register dumps as seeing the program itself is seldom helpful.

3) Care should be exercised if using the 'F' format option in that if a word is out of range for this format, the field is printed as all *'s, just as in Fortran.

4) Since the size of the dynamic portion of DIAG$ (the file into which all dumps are written) is fixed at 1,000 sectors, only approximately 2,500 total words may be dumped per execution.

9.2 Obtaining a Snapshot Dump Via SNAP$

To call SNAP$ the following instructions are necessary:

```
S       A0,PKT ADDR+2
L,U     A0,PKT ADDR
ER      SNAP$
```

Where PKT is a three word packet as follows:

WORD	35–32	32–17	17–0
0	snapshot identifier (6 characters field data)		
1	XAR	WORD-LENGTH	START-ADDR
2	former A0 contents		

The XAR field contains an octal number which specifies which sets of control registers to dump:

0	none	4	only X
1	only R	5	X and R
2	only A	6	X and A
3	R and A	7	all registers

As an example, suppose somewhere in your program you had:

```
      P        FORM    3,15,18
    LABEL        'FARBLE'
      P            7,0,0
      +              0
```

Then the instructions

```
      S        A0,LABEL+2
     L,U        A0,LABEL
     ER          SNAP$
```

would dump all the registers only.

The following system proc call generates, in sequence, the necessary three instructions, a J $+4 instruction, and the necessary three word packet, which is everything needed to accomplish a SNAP$ request:

```
L$SNAP    'snapshot-identifier',XAR,word-length, start-addr
```

Therefore, the following single line replaces the 3 lines used in the above example:

```
L$SNAP    'FARBLE',7,0,0
```

9.3 Obtaining a Post Mortem Dump

Only the most important aspects of the @PMD processor will be shown here. For further information consult U of M User Reference 70-01 or the Univac PRM.

The format of the @PMD statement is:

@PMD, options

The options are:

E - dump only if run termates in error.
C - dump only words that were changed during execution.
I - dump just the I bank portion of the program.
D - dump just the D bank portion of the program.

NOTE: If both the I and D options are used, the effect is the same as if neither was used (i.e., @PMD produces the same results as @PMD, ID).

9.4 Obtaining Dumps Via PDUMP

PDUMP is a Fortran subroutine that has been converted from the 7094 to the 1108. To call PDUMP from an assembly language program follow the subroutine linkage instructions given in section 3 of this manual.

A call to the PDUMP subprogram by the statement

$$\text{CALL} \quad \text{PDUMP}(A_1, B_1, F_1, \ldots, A_i, B_i, F_i, \ldots, A_n, B_n, F_n)$$

causes the indicated limits of core storage to be dumped and execution to be continued. An explanation of the arguments used with PDUMP are as follows:

1. A and B are variable data names that indicate the limits of core storage to be dumped; either A or B may represent upper or lower limits.
2. F_i is an integer indicating the dump format desired:

F = 0	dump in octal
1	dump as real
2	dump as integer
3	dump in octal with mnemonics

3. If no arguments are given, all of core storage is dumped in octal.
4. If the last argument F_n is omitted, it is assumed to be equal to 0 and the dump will be octal.

10. Glossary and Conventions

1.	INSTRUCTION FIELDS:	f - field contains function code designator (f) j - field contains operand qualifier or minor function code (j) a - field contains AXR-register designator, channel designator, or console keys designator (a) x - field contains index register designator (x) h - field contains index register modification designator (h) i - field contains indirect addressing designator (i) u - field contains address or operand designator (u)
2.	a-FIELD DESIGNATOR REFERENCES:	K_a = value of "a" designates console key C_a = value of "a" designates channel number A_a = value of "a" designates an A-register within set of A-registers X_a = value of "a" designates an X-register within set of X-registers R_a = value of "a" designates an R-register within set of R-registers
3.	AXR CONTROL REGISTER SETS:	A = A-registers = Accumulators X = X-registers = Index Registers R = R-registers = Special Purpose Registers
4.	X-REGISTER SUBSCRIPT	Subscript M = lower half of X-register (Modifier) Subscript I = upper half of X-register (Increment)
5.	ADDRESSES:	U = Program effective address (Relative Address) S = Main Storage address (Absolute Address) S_I = Main Storage address in I-Storage Area S_D = Main Storage address in D-Storage Area
6.	REGISTERS:	P = Program Address Register CR = Control Registers AR = Address Registers

7. SPECIAL SYMBOLS:

( ) = contents of specified register or storage address, subscripts indicate bit positions being considered, a prime (') superscript indicates the ones complement.

|( )| = Absolute value or magnitude.

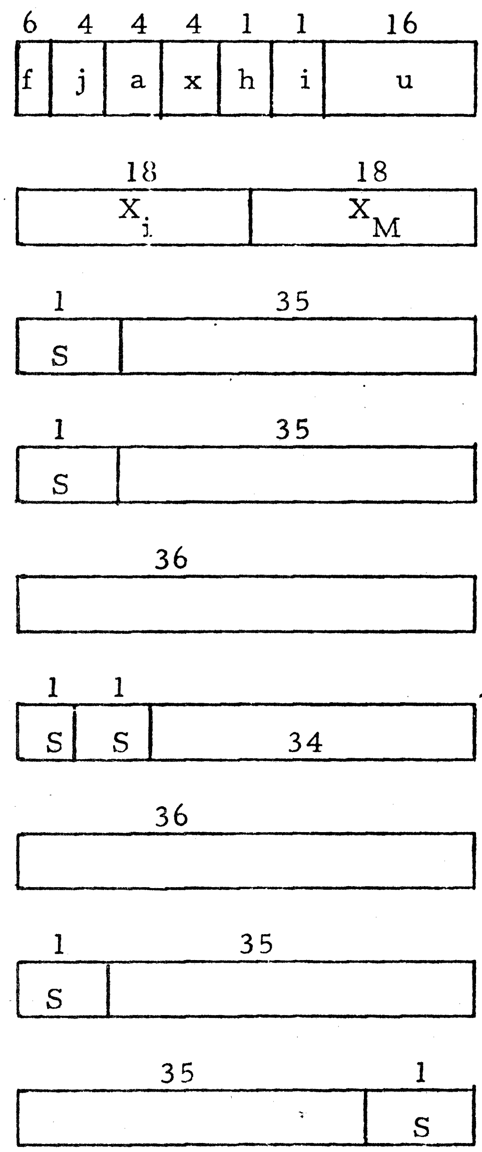
→ = direction of data flow or "goes to"

$\odot$ = logical AND function

$\oplus$ = logical OR function

$\overline{\oplus}$ = logical EXCLUSIVE OR

UNIVAC 1108 Processor Word Formats Numbers above segments indicate the number of bits in the segment.



Instruction Word

Index-Register Word

Single-Precision Fixed-Point Word

A or U

A+1 or U+1

Double-Precision Fixed-Point Word

A or U

A+1 or U+1

Fixed-Point Integer Multiply Result

A

A+1

Fixed-Point Fractional Multiply Result (Right-circular shift A+1 by one to align least-significant operand)

1	35
S	

Fixed-Point Multiply Single-Integer Result

35	18	17	0
S		S	0
	carry		carry

Add-Halves Word Format

35	24	23	12	11	0
S		S		S	
	c		c		c

Add-Thirds Word Format

1	8	27
S	C	M

Single-Precision Floating-Point Operand

A or A + 1

1	8	27
S	C_U	M_U

Single-Precision Floating-Point Result; $C_L = C_U - 27$.

1	8	27
	C_L	M_L

Word 2 contains un-normalized least significant result.

1	11	24
S	C	M_U

36
M_L

Double-Precision Floating-Point Operand or Result

APPENDIX A. CODE/SYMBOL RELATIONSHIPS

The following table shows the relationships between the octal computer codes, the 80 column card codes, and the characters or symbols represented by these codes.

COMPUTER CODE (OCTAL)	CARD CODE	CHARACTER	COMPUTER CODE (OCTAL)	CARD CODE	CHARACTER
00	7-8	@	40	12-4-8	)
01	12-5-8	[	41	11	—
02	11-5-8	]	42	12	+
03	12-7-8	#	43	12-6-8	<
04	11-7-8	Δ	44	3-8	=
05	(Blank)	(Space)	45	6-8	>
06	12-1	A	46	2-8	&
07	12-2	B	47	11-3-8	$
10	12-3	C	50	11-4-8	*
11	12-4	D	51	0-4-8	(
12	12-5	E	52	0-5-8	%
13	12-6	F	53	5-8	:
14	12-7	G	54	12-0	?
15	12-8	H	55	11-0	!
16	12-9	I	56	0-3-8	, (comma)
17	11-1	J	57	0-6-8	\
20	11-2	K	60	0	0
21	11-3	L	61	1	1
22	11-4	M	62	2	2
23	11-5	N	63	3	3
24	11-6	O	64	4	4
25	11-7	P	65	5	5
26	11-8	Q	66	6	6
27	11-9	R	67	7	7
30	0-2	S	70	8	8
31	0-3	T	71	9	9
32	0-4	U	72	4-8	' (apostrophe)
33	0-5	V	73	11-6-8	;
34	0-6	W	74	0-1	/
35	0-7	X	75	12-3-8	.
36	0-8	Y	76	0-7-8	□
37	0-9	Z	77	0-2-8	‡ (or stop)

APPENDIX B. INSTRUCTION REPERTOIRE

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

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

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

Function Code (Octal) f	j	Mnemonic	Instruction	Description②	1108 Execution Time in μsecs. ①	1106 Execution Time in μsecs. ⑥
36	0-17	DF	Divide Fractional	(A,A+1)÷(U)→A; REMAINDER→A+1	10.125	13.950
37	–	–	Illegal Code	Causes illegal instruction interrupt to address 241_8	–	–
40	0-17	OR	Logical OR	(A) OR (U)→A+1	.75	1.5
41	0-17	XOR	Logical Exclusive OR	(A) XOR (U)→A+1	.75	1.5
42	0-17	AND	Logical AND	(A) AND (U)→A+1	.75	1.5
43	0-17	MLU	Masked Load Upper	[(U) AND (R2)] OR [(A) AND (R2)'] →A+1	.75	1.5
44	0-17	TEP	Test Even Parity	Skip NI if (U) AND (A) have even parity	2.00 skip 1.25 NI	3.00 skip 2.166 NI
45	0-17	TOP	Test Odd Parity	Skip NI if (U) AND (A) have odd parity	2.00 skip 1.25 NI	3.00 skip 2.166 NI
46	0-17	LXI	Load X Increment	$(U)\to {X_a}_{35-18}$; ${X_a}_{17-0}$ unchanged	1.00	1.833
47	0-17	TLEM	Test Less Than or Equal To Modifier	Skip NI if $(U)\leq(X_a)_{17-0}$;	1.75 skip 1.00 NI	3.333 skip 1.833 NI
		TNGM	Test Not Greater Than Modifier	always $(X_a)_{17-0}+(X_a)_{35-18}\to {X_a}_{17-0}$		
50	0-17	TZ	Test Zero	Skip NI if (U)=±0	1.625 skip .875 NI	3.166 skip 1.666 NI
51	0-17	TNZ	Test Nonzero	Skip NI if (U)≠±0	1.625 skip .875 NI	3.166 skip 1.666 NI
52	0-17	TE	Test Equal	Skip NI if (U)=(A)	1.625 skip .875 NI	3.166 skip 1.666 NI
53	0-17	TNE	Test Not Equal	Skip NI if (U)≠(A)	1.625 skip .875 NI	3.166 skip 1.666 NI
54	0-17	TLE TNG	Test Less Than or Equal Test Not Greater	Skip NI if (U)≤(A)	1.625 skip .875 NI	3.166 skip 1.66 NI
55	0-17	TG	Test Greater	Skip NI if (U)>(A)	1.625 skip .875 NI	3.166 skip 1.66 NI
56	0-17	TW	Test Within Range	Skip NI if (A)<(U)≤(A+1)	1.75 Skip 1.00 NI	3.33 skip 1.66 NI
57	0-17	TNW	Test Not Within Range	Skip NI if (U)≤(A) or (U)>(A+1)	1.75 skip 1.00 NI	3.33 skip 1.66 NI
60	0-17	TP	Test Positive	Skip NI if $(U)_{35}=0$	1.50 skip .75 NI	3.0 skip 1.5 NI
61	0-17	TN	Test Negative	Skip NI if $(U)_{35}=1$	1.50 skip .75 NI	3.0 skip 1.5 NI
62	0-17	SE	Search Equal	Skip NI if (U)=(A), else repeat	2.25 + .75K always	3.5 + 1.5K always
63	0-17	SNE	Search Not Equal	Skip NI if (U)≠(A), else repeat	2.25 + .75K always	3.5 + 1.5K always
64	0-17	SLE SNG	Search Less Than or Equal Search Not Greater	Skip NI if (U)≤(A), else repeat	2.25 + .75K always	3.5 + 1.5K always
65	0-17	SG	Search Greater	Skip NI if (U)>(A), else repeat	2.25 + .75K always	3.5 + 1.5K always

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

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

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

Function Code (Octal) f	Function Code (Octal) j	Mnemonic	Instruction	Description [2]	1108 Execution Time in μsecs. [1]	1106 Execution Time in μsecs. [6]
72	02	JPS	Jump Positive and Shift	Jump to U if $(A)_{35}=0$; go to NI if $(A)_{35}=1$; always shift (A) left circularly one bit position.	1.50 jump .75 NI always	3.0 jump 1.5 NI always
72	03	JNS	Jump Negative and Shift	Jump to U if $(A)_{35}=1$; go to NI if $(A)_{35}=0$; always shift (A) left circularly one bit position	1.50 jump .75 NI always	3.0 jump 1.5 NI always
72	04	AH	Add Halves	$(A)_{35-18}+(U)_{35-18}\to A_{35-18}$; $(A)_{17-0}+(U)_{17-0}\to A_{17-0}$	.75 always	1.5 always
72	05	ANH	Add Negative Halves	$(A)_{35-18}-(U)_{35-18}\to A_{35-18}$; $(A)_{17-0}-(U)_{17-0}\to A_{17-0}$	.75 always	1.5 always
72	06	AT	Add Thirds	$(A)_{35-24}+(U)_{35-24}\to A_{35-24}$; $(A)_{23-12}+(U)_{23-12}\to A_{23-12}$; $(A)_{11-0}+(U)_{11-0}\to A_{11-0}$	.75 always	1.5 always
72	07	ANT	Add Negative Thirds	$(A)_{35-24}-(U)_{35-24}\to A_{35-24}$; $(A)_{23-12}-(U)_{23-12}\to A_{23-12}$; $(A)_{11-0}-(U)_{11-0}\to A_{11-0}$	.75 always	1.5 always
72	10	EX	Execute	Execute the instruction at U	.75 always	1.5 always
72	11	ER	Execute Return	Causes executive return interrupt to address 242_8	1.375 always	2.33 always
72	12	–	Illegal Code	Causes illegal instruction interrupt to address 241_8	–	–
72	13	PAIJ [9]	Prevent All I/O Interrupts and Jump	Prevent all I/O interrupts and jump to U	.75 always	1.5 always
72	14	SCN	Store Channel Number	If a=0: CHANNEL NUMBER $\to U_{3-0}$; If a=1: CHANNEL NUMBER $\to U_{3-0}$ and CPU NUMBER $\to U_{5-4}$	.75	1.5
72	15	LPS [9]	Load Processor State Register	(U)→Processor State Register	.75	1.5
72	16	LSL [9]	Load Storage Limits Register	(U)→SLR	.75	1.5
72	17	–	Illegal Code	Causes illegal instruction interrupt to address 241_8	–	–
73	00	SSC	Single Shift Circular	Shift (A) right circularly U places	.75 always	1.5 always
73	01	DSC	Double Shift Circular	Shift (A,A+1) right circularly U places	.875 always	1.5 always
73	02	SSL	Single Shift Logical	Shift (A) right U places; zerofill	.75 always	1.5 always
73	03	DSL	Double Shift Logical	Shift (A,A+1) right U places; zerofill	.875 always	1.5 always
73	04	SSA	Single Shift Algebraic	Shift (A) right U places; signfill	.75 always	1.5 always
73	05	DSA	Double Shift Algebraic	Shift (A,A+1) right U places; signfill	.875 always	1.666 always
73	06	LSC	Load Shift and Count	$(U)\to A$, shift (A) left circularly until $(A)_{35}\neq(A)_{34}$; NUMBER OF SHIFTS $\to A+1$	1.125	2.0

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

Function Code (Octal) f	j	Mnemonic	Instruction	Description (2)	1108 Execution Time in μsecs. (1)	1106 Execution Time in μsecs. (6)
73	07	DLSC	Double Load Shift and Count	(U,U+1)→A,A+1; shift (A,A+1) left circularly until $(A,A+1)_{71} \neq (A,A+1)_{70}$; NUMBER OF SHIFTS→A+2	2.125	3.830
73	10	LSSC	Left Single Shift Circular	Shift (A) left circularly U places	.75 always	1.5 always
73	11	LDSC	Left Double Shift Circular	Shift (A,A+1) left circularly U places	.875 always	1.666 always
73	12	LSSL	Left Single Shift Logical	Shift (A) left U places; zerofill	.75 always	1.5 always
73	13	LDSL	Left Double Shift Logical	Shift (A,A+1) left U places; zerofill	.875 always	1.666 always
73	14	III (9) (a=0 or 1)	Initiate Interprocessor Interrupt (1108 System only)	Initiate interprocessor interrupt	.75 always	-
		ALRM (7) ($a=10_8$)	Alarm	Turn on alarm	.75 always	1.5 always
		EDC (9) ($a=11_8$)	Enable Day Clock	Enable day clock	.75 always	1.5 always
		DDC (9) ($a=12_8$)	Disable Day Clock	Disable day clock	.75 always	1.5 always
73	15	SIL (9)	Select Interrupt Locations	(a)→MSR	.75 always	1.5 always
73	16	LCR (9) (a=0)	Load Channel Select Register	$(U)_{3-0}$→CSR	.875	1.666
		LLA (9) (a=1)	Load Last Address Register	$(U)_{2-0}$→LAR	.875	1.666
73	17	TS	Test and Set	If $(U)_{30}=1$, interrupt to address 244_8; if $(U)_{30}=0$, go to NI; always 01_8→U_{35-30}; $(U)_{29-0}$ unchanged	Alternate bank: 1.625 interrupt .875 NI	3.166
					Same bank: 2.0 interrupt 2.0 NI	1.666
74	00	JZ	Jump Zero	Jump to U if (A)=±0; go to NI if (A)≠±0	1.50 jump .75 NI always	3.0 jump 1.5 NI always
74	01	JNZ	Jump Nonzero	Jump to U if (A)≠±0; go to NI if (A)=±0	1.50 jump .75 NI always	3.0 jump 1.5 NI always
74	02	JP	Jump Positive	Jump to U if $(A)_{35}=0$; go to NI if $(A)_{35}=1$	1.50 jump .75 NI always	3.0 jump 1.5 NI always
74	03	JN	Jump Negative	Jump to U if $(A)_{35}=1$; go to NI if $(A)_{35}=0$	1.50 jump .75 NI always	3.0 jump 1.5 NI always
74	04	JK J	Jump Keys Jump	Jump to U if a=0 or if a=lit select jump indicator; go to NI if neither is true	.75 always	1.5 always
74	05	HKJ HJ	Half Keys and Jump Half Jump	Stop if a=0 or if |a AND lit select stop indicators|≠0; on restart or continuation, jump to U	.75 always	1.5 always

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

Function Code (Octal) f	j	Mnemonic	Instruction	Description [2]	1108 Execution Time in μsecs. [1]	1106 Execution Time in μsecs. [6]
74	06	NOP	No Operation	Proceed to next instruction	.75 always	1.5 always
74	07	AAI [9]	Allow All I/O Interrupts	Allow all I/O interrupts and jump to U	.75 always	1.5 always
74	10	JNB	Jump No Low Bit	Jump to U if $(A)_0=0$; go to NI if $(A)_0=1$	1.50 jump .75 NI always	3.0 jump 1.5 NI always
74	11	JB	Jump Low Bit	Jump to U if $(A)_0=1$; go to NI if $(A)_0=0$	1.50 jump .75 NI always	3.0 jump 1.5 NI always
74	12	JMGI	Jump Modifier Greater and Increment	Jump to U if $(X_a)_{17-0} > 0$; go to NI if $(X_a)_{17-0} \leq 0$; always $(X_a)_{17-0}$ + $(X_a)_{35-18} \rightarrow X_{a\,17-0}$	1.50 jump .75 NI always	3.166 jump 1.5 NI always
74	13	LMJ	Load Modifier and Jump	(P)−BASE ADDRESS MODIFIER [BI or BD] $\rightarrow X_{a\,17-0}$; Jump to U	.875 always	1.666 always
74	14	JO	Jump Overflow	Jump to U if D1 of PSR=1; go to NI if D1=0	1.50 jump .75 NI always	3.0 jump 1.5 NI always
74	15	JNO	Jump No Overflow	Jump to U if D1 of PSR=0; go to NI if D1=1	1.50 jump .75 NI always	3.0 jump 1.5 NI always
74	16	JC	Jump Carry	Jump to U if D0 of PSR=1; go to NI if D0=0	1.50 jump .75 NI always	3.0 jump 1.5 NI always
74	17	JNC	Jump No Carry	Jump to U if D0 of PSR=0; go to NI if D0=1	1.50 jump .75 NI always	3.0 jump 1.5 NI always
75	00	LIC [9]	Load Input Channel	For channel [a OR CSR]:(U)→IACR; set input active; clear input monitor	.75	1.5
75	01	LICM [9]	Load Input Channel and Monitor	For channel [a OR CSR]:(U)→IACR; set input active; set input monitor	.75	1.5
75	02	JIC [9]	Jump On Input Channel Busy	Jump to U if input active is set for channel [a OR CSR]; go to NI if input active is clear	.75 always	1.5 always
75	03	DIC [9]	Disconnect Input Channel	For channel [a OR CSR]: clear input active; clear input monitor	.75 always	1.5 always
75	04	LOC [9]	Load Output Channel	For channel [a OR CSR]:(U)→OACR; set output active; clear output monitor; clear external monitor (ISI only)	.75	1.5 always
75	05	LOCM [9]	Load Output Channel and Monitor	For channel [a OR CSR]:(U)→OACR; set output active; set output monitor; clear external function (ISI only)	.75	1.5
75	06	JOC [9]	Jump On Output Channel Busy	Jump to U if output active is set for channel [a OR CSR]; go to NI if output active is clear	.75 always	1.5 always

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

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

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

Function Code (Octal) f	j	Mnemonic	Instruction	Description②	1108 Execution Time in μsecs. ①	1106 Execution Time in μsecs. ⑥
76	11	DFAN	Double Precision Floating Add Negative	$(A,A+1)-(U,U+1)\rightarrow A,A+1$	2.625	4.5
76	12	DFM	Double Precision Floating Multiply	$(A,A+1)\cdot(U,U+1)\rightarrow A,A+1$	4.25	6.667
76	13	DFD	Double Precision Floating Divide	$(A,A+1)\div(U,U+1)\rightarrow A,A+1$	17.25⑤	24.0⑧
76	14	DFU	Double Load and Unpack Floating	$\lvert(U)\rvert_{34-24}\rightarrow A_{10-0}$, zerofill; $(U)_{23-0}\rightarrow A+1_{23-0}$, signfill; $(U+1)\rightarrow A+2$	1.50	3.0
76	15	DFP	Double Load and Convert To Floating	$(U)_{35}\rightarrow A+1_{35}$; [NORMALIZED $(U,U+1)]_{59-0}\rightarrow A+1_{23-0}$ and A+2; if $(U)_{35}=0, (A)_{10-0}\pm$ NORMALIZING COUNT $\rightarrow A+1_{34-24}$; if $(U)_{35}=1$, ones complement of [$(A)_{10-0}\pm$ NORMALIZING COUNT] $\rightarrow A+1_{34-24}$		
76	16	FEL	Floating Expand and Load	If $(U)_{35}=0$, $(U)_{35-27}+1600_8\rightarrow A_{35-24}$; if $(U)_{35}=1$, $(U)_{35-27}-1600_8\rightarrow A_{35-24}$; $(U)_{26-3}\rightarrow A_{23-0}$; $(U)_{2-0}\rightarrow A+1_{35-33}$; $(U)_{35}\rightarrow A+1_{32-0}$	1.00	1.833
76	17	FCL	Floating Compress and Load	If $(U)_{35}=0$, $(U)_{35-24}-1600_8\rightarrow A_{35-27}$; if $(U)_{35}=1$, $(U)_{35-24}+1600_8\rightarrow A_{35-27}$; $(U)_{23-0}\rightarrow A_{26-3}$; $(U+1)_{35-33}\rightarrow A_{2-0}$	1.625	3.167
77	0-17	–	Illegal Code	Causes illegal instruction interrupt to address 241_8	–	–

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

NOTES:

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

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

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

② NI stands for Next Instruction.

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

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

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

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

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

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

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

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

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

⑨ Instructions so marked are illegal in guard mode.

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

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

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

Table B-3

j - Determined Partial-Word Operations

S = Sign extension, where the sign is the leftmost bit of partial word defined by j.

j	PSR BIT 17	MNEMONICS	BITS FROM (U)→BIT POSITIONS IN ARITHMETIC SECTION	BITS FROM ARITHMETIC SECTION → BIT POSITION OF U
00	-	W or None	35-0 → 35-0	35-0 → 35-0
01	-	H2	17-0 → 17-0	17-0 → 17-0
02	-	H1	35-18 → 17-0	17-0 → 35-18
03	-	XH2	17-0 → S 17-0	17-0 → 17-0
04	0 1	XH1 Q2	35-18 → S 17-0 26-18 → 8-0	17-0 → 35-18 8-0 → 26-18
05	0 1	T3 Q4	11-0 → S 11-0 8-0 → 8-0	11-0 → 11-0 8-0 → 8-0
06	0 1	T2 Q3	23-12 → S 11-0 17-9 → 8-0	11-0 → 23-12 8-0 → 17-9
07	0 1	T1 Q1	35-24 → S 11-0 35-27 → 8-0	11-0 → 35-24 8-0 → 35-27
10	-	S6	5-0 → 5-0	5-0 → 5-0
11	-	S5	11-6 → 5-0	5-0 → 11-6
12	-	S4	17-12 → 5-0	5-0 → 17-12
13	-	S3	23-18 → 5-0	5-0 → 23-18
14	-	S2	29-24 → 5-0	5-0 → 29-24
15	-	S1	35-30 → 5-0	5-0 → 35-30
16	-	U	18 bits* → 17-0	NO TRANSFER
17	-	XU	18 bits* → S 17-0	NO TRANSFER

* If x = 0: [h,i,u] is transferred (Exception: all 1-bits yield + 0)

If $x \neq 0$: $u + (X_x)_{17-0}$ is transferred

APPENDIX C. ASSEMBLER ERROR FLAGS AND MESSAGES

C.1. ERROR FLAGS

C.1.1. R-Relocation

An R flag indicates that a relocatable item (usually a label) has been so used in an expression as to cause loss of its relocation properties.

C.1.2. E-Expression

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

C.1.3. T-Truncation

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

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

The form reference in line (1) is legitimate, but (2) would produce a T flag, since the value of A in this case is 000000777774_8 (a value with 18 significant bits), and the second field of form "G" is defined as four bits in length.

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

C.1.4. L-Level

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

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

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

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

C.1.5. D-Duplicate

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

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

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

C.1.6. I-Instruction

If the first subfield in the operation field of a symbolic line contains neither the name of a directive, nor an available procedure, nor a FORM reference, nor a mnemonic, an I flag is produced. A procedure is considered available only if it is in the procedure library (that is, the system relocatable library or a user's file), or if it has previously been encountered in the source program. With the current level of the assembler (0011A), whenever an I flag is produced by the assembler, the corresponding bad line of code is generated as a NØP instead of 0's as in previous versions.

C.1.7. U-Undefined

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

C.2. ERROR MESSAGES

1108 ASM Internal Error Abort

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

Abort Cannot Read PROC from Drum

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

Assembler Image

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

ASM Abort no Scratch File A0 XXXXX

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

Bad Procedure Read

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

Item Table Overflow

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

Line Number Sequence Errors

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

PARTBL Not Initialized

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

Procedure Sample Storage Overflow

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

ROR Internal Error Abort

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

TBLWR$ Internal Error Abort

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

QUICK REFERENCE FOR DECODING ASSEMBLY LISTINGS

27	00	13	00	0	000043	LX X11,I
f	j	a	x	h i	u	
function code and j designator from Appendix B Table B-2 and B-3 page 37-38					address of LOC	

a	A,X,or R register number
0	0
1	1
2	2
3	3
4	4
5	5
6	6
7	7
10	8
11	9
12	10
13	11
14	12
15	13
16	14
17	15

x	X register
0	X0
1	X1
2	X2
3	X3
4	X4
5	X5
6	X6
7	X7
10	X8
11	X9
12	X10
13	X11
14	A0=X12
15	A1=X13
16	A2=X14
17	A3=X15

hi	meaning	action
0	neither bit set	no indirection or incrementation
1	i bit set	indirect addressing
2	h bit set	index incrementation
3	h & i bits set	do both

APPENDIX E

```
ALPH,ISL MAIN
IVERSITY OF MARYLAND REENTRANT ALGORITHM LANGUAGE PROCESSOR, VERSION    44.22
CLE 00 COMPILED AT 09:12:00 ON THURSDAY, SEPTEMBER 2, 1971.
001          0101:        INTEGER A,B,C
002          0102:        A=3
003          0103:        B=2
004          0104: 3      CALL SUB(A,B,C,$99)
005          0105:        PRINT 10,C
006          0106: 10     FORMAT(' STANDARD RETURN C = ',I2)
007          0107:        A=11
008          0110:        GO TO 3
009          0111: 99     PRINT 20,C
010          0112: 20     FORMAT(' ABNORMAL RETURN C = ',I2)
011          0113:        END
```

ASSEMBLY CODE PRODUCED BY RALPH FOR THE SUBROUTINE CALL

```
                                          .
          0104:                           .      3      CALL SUB(A,B,C,$99)
                                          .
000011  74 13 13 00 0 000000    #00003    LMJ          X11,SUB
000012  000000000001                      +            A
000013  000000000002                      +            B
000014  000000000003                      +            C
000015  74 04 00 00 0 000030              J            #00099
000016  000104 000000                     +            000104,WBCK$
```

```
@ASM,IS SUB
ASM11D 09/02-09:12-(00)
  1.                                                  . THIS SUBROUTINE WILL ADD A & B IF A<=5 AND WILL MULTIPLY A BY 2 IF A>5
  2.                                                  . THE PROC <P> DETERMINES WHAT ACTION IS TO BE TAKEN BY THE SUBROUTINE
  3.                                                  . RETURN OF CONTROL TO THE CALLING PROGRAM IS MADE NORMALLY IF A<=5
  4.                                                  . AND ABNORMALLY (TO A STATEMENT LABEL IN THE MAIN PROGRAM) IF A>5
  5.                                                           AXR$                            . SET UP THE REGISTERS
  6.                                                  P.       PROC                            . DEFINE THE PROC
  7.                                                           TG,U      P(1,1),5              . IS 5 GREATER THAN P(1,1) ?
  8.                                                           J         P(1,2)                . 5 IS LESS THAN A, MULTIPLY A BY 2
  9.                                                           J         P(2,1)                . 5 IS GREATER THAN A, ADD A & B
 10.                                                           END                             . END OF THE PROC
 11.     00    000000  10 00 00 13 1 000000           SUB.     LA        A0,*0,X11             .  LOAD  A0 WITH A
 12.           000001  55 16 00 00 000005                      P         A0,NOTNUM ALPHA.       PROC CALL
               000002  74 04 00 00 0 000004
               000003  74 04 00 00 0 000007
 13.           000004  31 16 00 00 000002           NOTNUM   MSI,U     A0,2                  . MULTIPLY A BY 2
 14.           000005  01 00 00 13 1 000002                    SA        A0,*2,X11             .  STORE RESULT INTO C
 15.           000006  74 04 00 13 0 000003                    J         3,X11                 . ABNORMAL RETURN
 16.           000007  10 00 01 13 1 000001           ALPHA    LA        A1,*1,X11             . LOAD A1 WITH B
 17.           000010  14 00 00 00 0 000015                    AA        A0,A1                 . ADD A AND B
 18.           000011  01 00 00 13 1 000002                    SA        A0,*2,X11             . STORE RESULT INTO C
 19.           000012  74 04 00 13 0 000005                    J         5,X11                 . NORMAL RETURN
 20.                                                           END                             . END OF THE SUBROUTINE

END ASM.       719 MSEC.


@XQT
MAP  22C-09/02-09:12
START=007443, PROG SIZE(I/D)=3391/2291


END MAP  1108 MSEC.

 STANDARD RETURN C =  5
 ABNORMAL RETURN C = 22


NORMAL EXIT.  EXECUTION TIME:              2 MILLISECONDS.


@FIN
```