INTER-LINK SYSTEMS
10601 Highway 85, Suite 212
Cupertino, California 95014
408/257-7165

gri-909

basic assembler

GRI Computer Corporation
320 NEEDHAM STREET, NEWTON, MASSACHUSETTS 02164

Price $3.75

GRI-909

Basic Assembler

GRI Computer Corporation, 320 Needham Street, Newton, Massachusetts 02164

71-44-001-B
0371

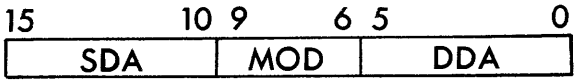
CONTENTS

APPENDICES

CHAPTER ONE

THE BASIC ASSEMBLER

The GRI-909 Direct Function Processor is a highly modular, general-purpose digital computer. Its programmability and functional architecture enable the solution of a wide variety of system control and processing problems. An object program to be run on the GRI-909 consists of a sequence of binary coded machine instructions and data to be operated upon. GRI-909 basic machine instructions are described by a single internal format:



where: SDA is the source device address,
MOD contains modifier or function information, and
DDA is the destination device address.

In effect, information in the form of either data or control signals is transmitted from the source device specified by SDA to the destination device at DDA. The qualities of the transmission and/or the end result of the instruction is influenced by the specification of MOD. A complete machine instruction consists of either 1) a basic instruction in the above format or 2) a basic instruction followed by a word containing a memory address or data for the instruction.

The assembly language supported by the Basic Assembler is oriented to the functional organization of the computer itself. The foregoing SDA MOD DDA format is employed throughout this manual to illustrate the relationship between an assembly language instruction and its equivalent in the object program.

1.1 INTRODUCTION

The Basic Assembler is an indispensable aid to the process of preparing binary object programs for the GRI-909 Computer. The Assembler enables the writing of programs in a terse and easily understood symbolic language, called the assembly language. The symbolic form of a program is called the source program and consists of a meaningful sequence of assembly language statements. The key

item in any statement is a mnemonic code which identifies the statement type. For instance:

1) The code RM denotes the machine instruction, register-to-memory data transmission,
2) The code ASC enables the insertion of ASCII text into the program as data, and
3) The code END denotes the end of a program - this code represents a directive to the assembler itself and does not cause the generation of binary information for the object program.

The assembler interprets each such code and either generates the appropriate binary object information or performs the implied assembler function.

A further advantage to assembly language programming is that instructions and data values may be labeled with user symbols - this enables them to be easily referenced from other points in the source program. The assembler performs the tedious chore of associating actual memory addresses with instructions and data. Lastly, complex data values to be inserted into the object program may be represented by expressions which are evaluated by the assembler.

The assembler, then, translates an assembly language source program into its equivalent binary object form. During this process the assembler maintains an internal variable, called the LOCATION COUNTER, which continuously reflects the object program memory address for which source statements are being assembled. As each statement is read the LOCATION COUNTER is automatically updated by the number of machine words that will be occupied by the assembled statement. As user symbols are encountered, they are added to a symbol table. When a symbol is encountered as a label for an instruction or data value, it is assigned the current value of the LOCATION COUNTER. Since such a symbol may be used in a program before its actual value is determined, the assembler must read the source program once (PASS 1) in order to define all symbols introduced by the user. The source program is read again (PASS 2) and the object program is generated. An optional third pass yields an assembly listing as described in the next section.

Source programs on paper tape are prepared using the Source Text Editor (Manual #72-44-001). The Text Editor is also used to generate new versions of source programs. When reading a source program from paper tape the assembler follows the conventions as presented in section 1.2, "SYSTEM CONVENTIONS" in the Text Editor.

1.2 ASSEMBLER OUTPUT

Pass 2 of the assembler reads the source program, and using the table of symbols defined during pass 1, generates the corresponding object program. The basic version of the assembler punches the object program onto paper tape. The assembler can easily be modified to accommodate the other standard peripheral devices, however. The object program is subsequently loaded into the GRI-909 via the absolute loader (%ALD).

The optional third assembler pass generates an assembly listing. This listing contains the source program statements and the object program data generated from each statement. Although source input statements are of free-form, the assembler separates major source fields to enhance the legibility of the listing. The program segment in the following listing example computes the sum and difference of X1 and X2.

```
LINE #  LOCATION  DATA          LABEL   INSTRUCTION  OPERANDS     COMMENTS
001 00100 02 0000 13 COMP:    FOA   ADD        ;SELECT ADD FUNCTION
002 00101 06 0000 11          MR    X1,AX
    00102 000203
003 00103 06 0000 12          MR    X2,AY
    00104 000204
004 00105 13 0000 06          RM    AO,Y1      ;STORE SUM
    00106 000201
005 00107 12 0110 12          RSC   AY,P1      ;TWO'S COMP X2
006 00110 13 0000 06          RM    AO,Y2      ;STORE DIFF
    00111 000202
```

Before the line number there will appear one or two single-character codes if the assembler detected an error when interpreting the statement. The possible error codes and their meanings are:

M - Multiply defined symbol; the statement contains a label symbol that has been associated with more than one memory location.

U - Undefined symbol; the statement contains a symbol that has not been assigned a value.

S - Syntax error; the statement is not formed according to the rules as defined in this manual. The assembler generates two zero words as object output in this case.

D - Decimal digit in octal field; a numeric constant contains an 8 or 9 and the assembler's radix is set to octal.

E - Expression error; an expression in the operands field is invalid.

> - Too many expressions; the general operands field is a comma-separated list of expressions - in this case, there were more such expressions than required for the instruction.

< - Too few expressions in operands field.

V - Symbol table overflow; the number of user-introduced symbols has exceeded the capacity of the assembler's symbol table.

The line number is a decimal sequence number generated by the assembler. For paper tape input, this number corresponds exactly to the implicit line number in the Source Text Editor buffer if the source program were to be updated using the Editor.

The location field contains five octal digits representing the memory location for which the associated data is being assembled. The generated data is presented in one of two formats:

a) Machine instructions are printed as two octal digits (bits 15-10), four binary digits (bits 9-6) and two octal digits (bits 5-0). These sub-fields correspond to the SDA, MOD and DDA portions of an instruction word.

b) Other data is printed as 6 octal digits where the first digit may be only 0 or 1.

PASS 3 of the assembler also generates a listing of user-introduced symbols and their assigned values. A single symbol and its octal value appears on each line. The error code U (above) is printed before a symbol if appropriate.

CHAPTER TWO

LANGUAGE ELEMENTS

This chapter describes the basic elements that are used to form assembly language source statements. Throughout this manual the following notational conventions will be employed when presenting general forms of language elements:

[ ] Brackets - used to contain an optional item. The statement may be written with or without the item - generally, the meaning of the statement is changed when such an item is omitted.

{ } Braces - used to contain alternate items. These items will be arranged vertically within the braces - the statement must include one, and only one, of the alternate items.

... Ellipsis - used to denote permissible repetition of the immediately preceding language element.

When braces are enclosed within brackets, then either the entire form in brackets is omitted or the form is included with the appropriate alternate item selected.

2.1 CHARACTER SET

The GRI-909 basic assembler processes source statements composed of 8-bit ASCII characters, and recognizes two distinct categories of characters: general usage characters and reserved characters.

General usage characters are used to form symbols and simple numeric constants:

CHARACTER	EXTERNAL	INTERNAL
Alphabetics	A through Z	301 through 332
Numerics	0 through 9	260 through 271
Dollar Sign	$	244
Percent Sign	%	245
At Sign	@	300

Reserved characters are used to impart special meanings to the assembler, or to separate or delimit certain language elements:

CHARACTER	EXTERNAL	INTERNAL	FUNCTION
Carriage-return		215	Delimits source line.
Exclamation Point	!	241	Denotes logical "OR".
Ampersand	&	246	Denotes logical "AND".
Plus Sign	+	253	Denotes Addition.
Comma	,	254	Separates machine instruction operands or general list elements.
Minus Sign	-	255	Denotes Subtraction.
Period	.	256	Represents the assembler's location counter.
Colon	:	272	Separates a label from the rest of the statement.
Semi-colon	;	273	Separates comments from the rest of the statement.
Equals Sign	=	275	Separates a parameter symbol from the expression denoting its assigned value.
Back-arrow	←	337	Causes the first previous input character not a back-arrow to be ignored by the assembler.

CHARACTER	EXTERNAL	INTERNAL	FUNCTION
Block mark		375 or 233	Separates blocks of source statements. Valid only between lines - causes the assembler to reset the listing line number to one (1).
Rubout		377	Causes the previous portion of the input line to be ignored by the assembler.
Space		240	General delimiter.

NOTE: Although the assembler recognizes only 8-bit characters internally the source tape input may be in either 8-bit or even-parity code since the text input routine logically OR's the high-order bit into each character read.

2.2 SYMBOLS

Assembly language symbols are used to represent memory addresses, device or operator addresses, machine or assembler instructions, and simple numeric values. Pre-defined symbols in the assembler's symbol table have mnemonic value: for instance, the symbol RMID represents the machine instruction Register-to-Memory-Immediate-Deferred. User symbols, as defined in the source program, represent either a statement label (2.3) or an assembly parameter (2.4). In order to enhance the utility of assembly listings, the user should attempt to define his symbols with mnemonic value as well.

A symbol consists of one or more non-blank general usage characters, the first of which must not be numeric. Since only the first five characters are stored in the symbol table, symbols of greater length must be unique in the first five characters.

The following symbols are valid and could be used as a label or as a parameter:

START

LOOP

N23@

PARA11

PARA21

The following symbols are invalid for the reasons given:

8ABC	First character numeric
LOOP*	Invalid character (*)
GO;TO	reserved character (;)
AB LE	Embedded blank
PARAM1	Not unique in the first five characters
PARAM2	

The user is further cautioned not to define symbols identical to any of those in the standard assembler symbol table (Appendix C) unless it is intended to alter their meaning.

2.3 LABELS

A statement label is defined or established by the occurrence of

Symbol:

(a symbol followed by the reserved character:) as the first element of an assembly language source statement. The assembler assigns the current value of the location counter to this label - this will be the memory address of the first word assembled from the statement with which the label is associated.

A label is used to symbolically reference a specific instruction or data word from other points in the program. Therefore, a given label must not be re-defined within the same program - an attempt to associate a label with two or more different memory addresses will cause an error code to be printed on the assembly listing.

EXAMPLES:

```
TYPE:    RR AX,TTO
TABLE:   WRD - 144, - 12, - 1
```

2.4 PARAMETERS

An assembly parameter is defined by the occurrence of

Symbol = e

(a symbol followed by the reserved character =, followed by an expression e) as an assembly language statement. Expressions are defined in section 2.6. The value of the parameter will be the assembled value of the expression with which it is associated.

A parameter is used to represent device addresses, function generation pulse codes, or function testing status codes (See chapter 3). A parameter may also be used to represent a numeric value to be used in the formation of other expressions. Note that no object code is generated by a parameter assignment statement - the statement merely causes a numeric value to be assigned to the parameter symbol.

An assembly parameter may be redefined within the same program. If a parameter does take on more than one value, then its initial value must be established in the source program before it is first used to symbolically reference a numeric value.

EXAMPLES:

```
TEN = 10
CR = 215
DIFF = A-B
```

The expression used to specify the value of a parameter must be fully resolvable by at least the end of pass 1 (so that it will have the correct value during passes 2 and 3). In other words, the value of any symbol in the expression must be established within at most one forward reference. For example, in the sequence:

```
A = B + 5
B = 22
```

the value of A cannot be established when the statement is first encountered during pass 1 since B is not yet defined. Nevertheless, when the definition of A is encountered during pass 2, it will be assigned the correct value (27). On the other hand, in the sequence:

```
A = B + 5
B = C - 2
C = 24
```

the value of A cannot be established until its definition is encountered during pass 3 of the assembler. Since the assembler assigns a value of zero (0) to undefined symbols, the values assigned to the symbols in this example when their defining statements are encountered will be as follows:

	A	B	C
Pass 1	5	-2	24
Pass 2	3	22	24
Pass 3	27	22	24

2.5 CONSTANTS

A simple constant is represented by one or more successive numeric characters. The character string is converted into its equivalent binary value according to the setting of the assembler's radix, which may be either decimal or octal (see section 4.2). The range of a constant, so as not to arithmetically overflow out of the fifteen magnitude bits of a signed machine word, is 0 to 32767 decimal or 0 to 77777 octal.

If a stand-alone constant is to be treated as an unsigned (magnitude only) entity, the upper limits may then be 65535 decimal or 177777 octal.

2.6 EXPRESSIONS

Compound numeric values may be formed in instruction fields or data words by arithmetically and/or logically combining simple values in an assembly language expression. An expression consists of a numeric operand or a series of operands separated by arithmetic and/or logical operators, where the first such operand may be preceded by an arithmetic operator (leading sign). Any given operand may be one of the following:

Label
Parameter
Constant
. (representing the assembler's location counter)

and the permissible operators are:

+	Denotes addition
-	Denotes Subtraction
&	Denotes logical "AND"
!	Denotes logical "OR"
Space	Used to imply logical "OR"

A general expression, e, is assembled into a 16-bit value. The formal definition of e is:

$$\left[\pm\right] \left\{\begin{array}{l}\text{Label}\\ \text{Parameter}\\ \text{Constant}\\ .\end{array}\right\} \left[\left\{\begin{array}{l}+\\ -\\ \&\\ !\\ \text{Space}\end{array}\right\} \left\{\begin{array}{l}\text{Label}\\ \text{Parameter}\\ \text{Constant}\\ .\end{array}\right\}\right] \ldots$$

An expression is evaluated in a simple left-to-right scan: no priorities are assigned to the operators.

EXAMPLES:

```
15
-237
A + 5
. + B - 3
VAL1!VAL2&VAL3
```

2.7 COMMENTS

User comments may be inserted in any line of the source program by separating them from the rest of the line by the special character ; (semi-colon). A comment must either be the last element of a source line or it must be the first and only element.

EXAMPLES:

```
RSC AX, P1    ; NEGATE AX      (last element)
; CONVERSION ROUTINE           (only element)
```

Only as much of a comment as will fit will appear on the assembly listing - the remainder of the comment, if any, will be ignored. There are no special rules regarding the characters, or their spacing, that may be contained in the body of a comment, except that:

1) a carriage-return terminates the source line
2) the back-arrow and rubout characters perform the functions presented in section 2.1

2.8 STATEMENTS

A source program statement (a source line) is a meaningful arrangement of basic language elements and is terminated by a carriage-return. A statement may contain no more than 80 characters, including spaces (blanks). An assembly language statement may take on one of the following general forms:

```
SYMBOL:  INSTRUCTION OPERANDS ; COMMENTS
SYMBOL:  INSTRUCTION OPERANDS
```

```
SYMBOL:   INSTRUCTION              ; COMMENTS
SYMBOL:   INSTRUCTION
          INSTRUCTION OPERANDS     ; COMMENTS
          INSTRUCTION OPERANDS
          INSTRUCTION              ; COMMENTS
          INSTRUCTION
SYMBOL = e  ; COMMENTS
SYMBOL = e
;  COMMENTS
;
```

where an INSTRUCTION is a machine instruction, an assembler instruction, or a pseudo instruction (described in Chapters 3, 4 and 5 respectively) and OPERANDS is a comma - separated list of expressions.

Other than the order of the major elements, as shown above, there are no formatting requirements imposed upon a source statement. The assembler isolates the major elements of a free-form source statement and arranges them in columns on the assembly listing.

The most basic elements (symbols) must, however, be separated or delimited from each other. Since symbols consist solely of general usage characters (2.1), a statement such as

```
VALU=V1+V2-V3;DEFINE VALUE
```

is easily understood by the assembler. Therefore, the main rule to be observed when preparing source statements is:

> When any two successive symbols are not separated by a reserved character, then they must be separated by at least one space.

C H A P T E R T H R E E

MACHINE INSTRUCTIONS

Although all basic GRI-909 machine instructions have the same format -- SDA MOD DDA - (See Chapter 1), the assembler distinguishes four general classes of instructions as follows:

Function Generation -- Control pulses specified by MOD are transmitted to the named destination device; the unique combination of MOD and DDA defines the function to be performed.

Function Testing -- Status indicators associated with the named source device are sensed and program flow is altered if the test specified by MOD is true; flow alteration, if any, consists of a skip over the next two memory words.

Data Testing -- Data in the named source device register is tested and program flow is altered if the test specified by MOD is true; flow alteration, if any, consists of an absolute transfer (jump) to some new location specified by the instruction.

Data Transmission -- Data is transmitted from the named source device to the named destination device; binary modifications to data in transit and, for memory-reference instructions, addressing modes are specified by MOD.

An assembly language machine instruction consists of a mnemonic followed by one to three expressions separated by commas. The expressions in the operands portion of the instruction are arranged according to the SDA MOD DDA order, left to right. These expressions provide values to be assembled into specific fields of the complete machine instruction. For two-word instructions, either the leftmost or the rightmost expression (as implied by the mnemonic) is assembled into the second word.

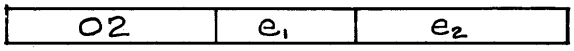
NOTE -- The value of any given expression to be packed into n bits of an instruction is treated modulo 2^n.

In order to render assembly language program listings more meaningful and to minimize the amount of writing necessary for the specification of instructions, the assembler's complement of mnemonics provides useful subdivisions within each of the four machine instruction classes. These classes and their subdivisions are described in the following sections. Machine word layouts presented with general forms detail the contribution of statement components to the assembled instruction.

3.1 FUNCTION GENERATION

Function generation instructions cause up to four pulses to be transmitted in parallel to controllable destination devices. A general function generation instruction is of the form:

FO e_1, e_2



where bits 9-6 of MOD correspond to the four machine pulse control lines.

EXAMPLES:

```
FO   1,77          ; START TTI READER
FO  11,76          ; CLEAR FLAG, START HSR
FO   2,0           ; SET LINK
FO   4,13          ; SELECT ARITH. OPERATOR "AND"
```

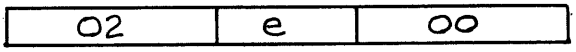
Using standard assembler symbols (or user-defined symbols) for function codes and device names, the previous examples might be written:

```
FO  STRT, TTI
FO  CLIF STRT, HSR
FO  STL,0
FO  AND, AO
```

The assembler provides mnemonics which imply a specific destination. These are of the form:

- machine (control logic)

FOM e

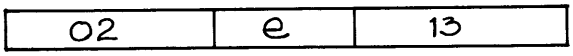


- interrupt control

FOI e

- arithmetic operator

FOA e



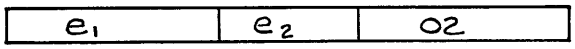
EXAMPLES:

```
FOM  STL        ; SET LINK
FOM  HLT        ; HALT MACHINE
FOI  ICO        ; INTERRUPT CONTROL-ON
FOA  ADD        ; SELECT AO "ADD"
FOA  AND        ; SELECT AO "AND"
```

3.2 FUNCTION TESTING

Function testing instructions enable the user to alter program flow based on the setting of status indicators associated with a given device. If the specified test is true, a skip over the next two words is performed. A general function testing instruction is of the form:

SF e_1,e_2



where MOD (9-7) correspond to the three machine sensing lines, and MOD (6) is interpreted as follows:

0 -- Skip on the "OR" of the truth of the selected indicators.

1 -- Skip on the "AND" of the falsity of the selected indicators.

EXAMPLES:

```
SF   77,2       ; SKIP IF TTY OUTPUT READY
SF   76,3       ; SKIP IF HSP NOT READY
SF    0,2       ; BUS OVERFLOW SET?
SF   13,2       ; SKIP AO OVERFLOW
```

Using standard symbols, the above examples are written:

```
SF   TTI,ORDY
SF   HSP, NOT ORDY
SF   0, BOV
SF   AO, AOV
```

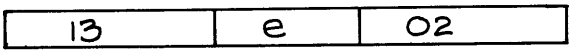
The assembler provides mnemonics which imply a specific source. These are of the form:

- machine

SFM e

- arithmetic operator

SFA e



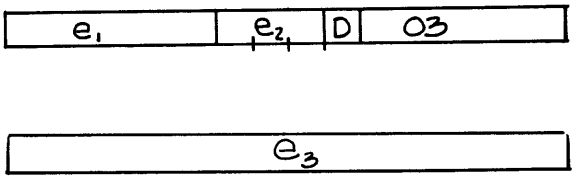
EXAMPLES:

```
SFM  BOV    ; BUS OVERFLOW SET?
SFM  NOT BOV LNK
SFA  AOV    ; SKIP AO OVERFLOW
```

3.3 DATA TESTING

Data testing instructions enable the user to alter program flow based on the value of data residing in a given device. The data is tested relative to algebraic zero. If the specified test is true, a jump is performed to some new program location. A general data testing instruction is of the form:

JC[D] e_1,e_2,e_3



where MOD (9-8) specify test conditions

- if MOD (9) = 1, test for less than zero
- if MOD (8) = 1, test for equal to zero,

MOD (7) is interpreted as follows:

0- jump on the "OR" of the truth of the selected test conditions.

1- jump on the "AND" of the falsity of the selected test conditions,

and the optional D, if included, sets MOD (6) which selects the deferred addressing mode.

Normally, the expression e_2 will consist solely of one of the assembler's predefined test codes:

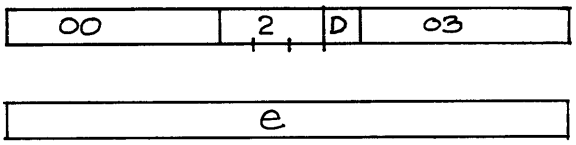
CODE	VALUE	CONDITION
ETZ	2	Equal to zero
NEZ	3	Not equal to zero
LTZ	4	Less than zero
GEZ	5	Greater than or equal to zero
LEZ	6	Less than or equal to zero
GTZ	7	Greater than zero

EXAMPLES:

```
JC    AX,GEZ,LOOP+5
JC    TTI,ETZ,AGAIN
JCD   AO,LTZ,SUB+1
JC    0,ETZ,.-7
```

The last example is an unconditional jump, since device zero is a source of a zero data word. The assembler provides a mnemonic for unconditional jumps:

JU[D] e



EXAMPLES:

```
JU   GO        ; JUMP TO GO
JUD  ADDR      ; JUMP DEFERRED THRU ADDR
```

3.4 DATA TRANSMISSION

Any machine instruction not specifically falling into one of the aforementioned three classes implies the transmission of data from a source device, through the Bus Modifier, to a destination device. Programmable data paths in the Bus Modifier enable the selection of binary modifications to data as it passes between the source and destination devices. The operands portion of every assembly language data transmission instruction contains an optional expression which, if included, is assembled into MOD (9-8). The modifications that can be selected by MOD (9-8) and the standard codes that may be used to invoke them are:

P1 - Increment (add one)

L1 - Shift left one bit

R1 - Shift right one bit

Only one of the above modifications may be selected in any given data transmission instruction. When data is incremented (P1), the bus overflow indicator is set if, and only if, the source data was equal to -1 (all ones). If such overflow did not occur, then the overflow indicator will be cleared. After a transmission through the incrementing path, the status of the bus overflow indicator can be sensed with a SFM [NOT] BOV. Data is shifted (L1 or R1) circularly through a one-bit link in the Bus Modifier. After any shift, the new status of the Link may be sensed with a SFM [NOT] LNK. If it is desired to shift a zero (or a one) into the word being transmitted, the pre-transmission state of the Link may be ensured by a FOM CLL (FOM STL).

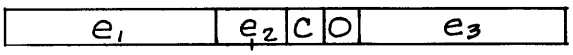
The zero or null device address may be used in data transmission instructions. When used as a source, it provides a zero data word which is transmitted, with or without modification, to the named destination. When used as a destination, the source data may be transmitted and Bus Modifier status indicators subsequently tested without modifying the source data itself or replacing the contents of some other device register.

Any data transmission instruction may be used to effect an absolute transfer of program control (jump) by transmitting a memory address to the computer's sequence counter (SC). Note, however, that if the transmission instruction is a one or a two-cycle instruction, then one less than the desired jump address must be transmitted.

3.4.1 NON-MEMORY TRANSMISSION

These instructions enable the transmission of data between non-memory registers in system devices, and have the general form:

RR[C] $e_1[,e_2],e_3$



where the optional C, if included, sets MOD (7) -- this bit, available only in non-memory transmissions, selects the ones complementation of data prior to another modification selected (if any).

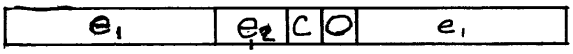
EXAMPLES:

```
RR    0,AX          ; CLEAR AX
RRC   AO,P1,AX      ;2's COMP OF AO TO AX
```

```
RR   TTI,TTO        ;TTI TO TTO
RR   AX,P1,AX       ;INCREMENT AX
```

This last example involves the transmission of a register to itself. The assembler provides the shorter form

RS[C] e_1[,e_2]



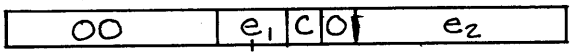
EXAMPLES:

```
RS   AX,P1          ;INCREMENT AX
RSC  AX             ;1's COMP AX
RS   AY,L1          ;SHIFT AY LEFT
```

NOTE - Not all system devices may be <u>both</u> a source and destination for data. For instance, AO, TTI, and HSR are source only, while TTO and HSP are destination only.

As previously noted, registers may be cleared by transmitting to them from the zero address. A further mnemonic is provided to facilitate this.

ZR[C] [e_1,] e_2



EXAMPLES:

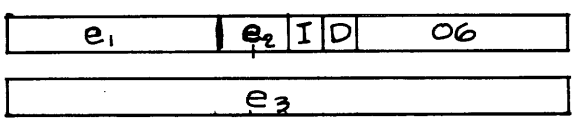
```
ZR   AX             ;AX=0
ZR   P1,HSP         ;PUNCH A ONE
ZRC  AY             ;SET AY TO -1
```

3.4.2 MEMORY REFERENCE TRANSMISSION

These instructions enable the transmission of data either between a device register and a memory location or from a given memory location to itself. In either case, the optional characters I and D in the instruction mnemonic cause the selection of the immediate and deferred addressing modes respectively.

Registers may be stored in memory using the general form:

RM[I] [D] e_1[,e_2],e_3

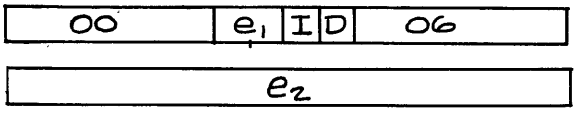


EXAMPLES:

```
RM   AX,SAVE1
RM   AO,P1,Z1+5      ;STORE OUTPUT+1
RMI  TRP,0           ;STORE TRAP IMMEDIATE
RMD  AX,ADDR
```

The clearing of memory locations is facilitated by the form:

ZM[I] [D] [e_1,]e_2

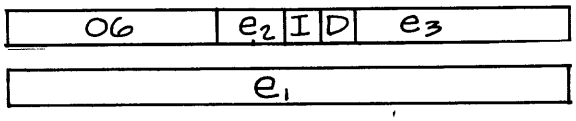


EXAMPLES:

```
ZM   COUNT           ;CLEAR COUNTER
ZM   P1,SW1          ;SET SWITCH
```

Registers may be loaded from memory using the general form:

MR[I][D] e_1[,e_2],e_3



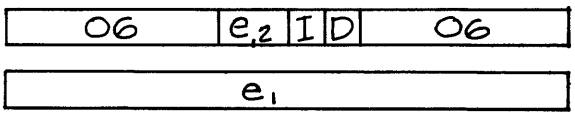
EXAMPLES:

```
MR   SAVE1,AX        ;RESTORE AX
MRI  215,TTO         ;TYPE CARRIAGE-RETURN
MRD  A,L1,AX
MRI  -1,TRP          ;TRAP=-1
```

NOTE - The last example has SDA=06 (memory) and DDA=03 (TRAP). This is the only instance where DDA=03 does not denote a data test instruction.

Memory locations may be modified through use of the general form:

MS[I][D] e_1[,e_2]



EXAMPLES:

```
MS   COUNT,P1        ;INCREMENT COUNTER
MSI  0,P1            ;INCR 2nd WORD OF INSTRUCTION
MS   MULTP,R1        ;ROTATE MULTIPLIER
```

CHAPTER FOUR

ASSEMBLER INSTRUCTIONS

This chapter describes assembly language instructions that either enable the insertion of data into the object program or merely act as directives to the assembler during the assembly process.

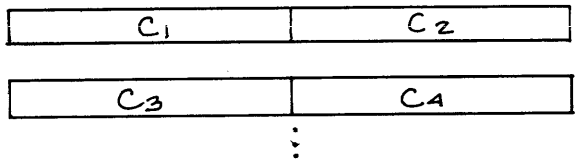
4.1 DATA DEFINITION

The following instructions enable the insertion of data into the object program.

4.1.1 TEXT

Consecutive characters of ASCII text are assembled into an object program using the form:

ASC $dc_1c_2c_3c_4\ldots c_nd$



where d, the delimiter, is the first non-blank character after the instruction mnemonic - the rightmost d must be the next character identical to the delimiter, and the c_i are text characters.

The text delimited by the d's is assembled into consecutive words, two ASCII characters per word, as shown. If the text contains an odd number of characters, the rightmost 8 bits of the last word assembled will be set to zero. Text characters may be drawn from other than the general usage or reserved character sets. The reserved characters carriage-return, back-arrow and rubout always perform their usual functions - See section 2.1. Hence, a carriage return cannot be used within the delimiters of an ASC statement.

EXAMPLES:

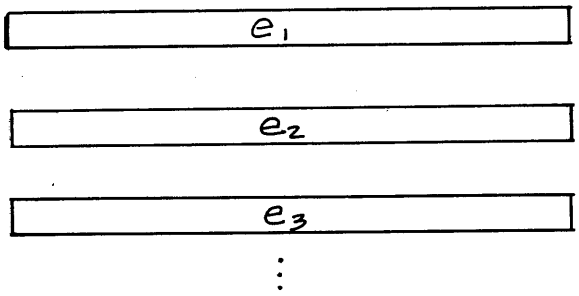
```
MSG:    ASC  /MOUNT NEXT TAPE/
        ASC  'A/B=LIM1'
ALRM1:  ASC  .ALARM '1'.
```

A label associated with an ASC instruction may be used to reference the first word assembled from the text.

4.1.2 WORD VALUES

Full 16 bit values of assembly language expressions may be assembled into consecutive words of the object program using the form:

WRD $e_1[,e_2]...$



The values of one or more expressions are assembled into the corresponding number of consecutive words.

EXAMPLES:

```
TABLE: WRD 1750,144,12    ;POWERS OF 10
COUNT: WRD 0
       WRD .+A-15
```

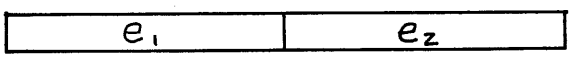
A label associated with a WRD instruction may be used to reference the first word assembled therefrom.

NOTE - If the assembler location counter symbol (.) is encountered in any expression in the list following WRD, its value will be the address of the word into which that expression is to be assembled.

4.1.3 PACKED BYTES

A pair of character or expression values may be assembled into the left and right halves of a word by using the form:

PKB e_1,e_2



EXAMPLES:

```
PKB  17,31
PKB  A+1,A-1
PKB  215,212
```

The last example packed a carriage-return and a line-feed character into a single word. Since the carriage-return cannot be included in the definition of a message (see ASC, above), it is often useful to follow the message with the foregoing packed character pair. Alternatively, one could write

```
CR=215
LF=212
PKB CR,LF
```

4.2 RADIX

The assembler converts constants (2.5) to their equivalent binary value according to the setting of an assembler variable, called the RADIX. The statement

OCTAL

causes the assembler to interpret subsequently encountered constants as octal numbers. The statement

DECIM[AL]

requests the assembler to interpret constants as decimal numbers.

The assembler's RADIX is initialized to OCTAL at the beginning of <u>each</u> pass. Any setting of RADIX by the user remains in effect until either the RADIX is reset or the pass is completed.

4.3 SET LOCATION

The assembler's LOCATION COUNTER, which continuously reflects the memory address into which a source program statement is being assembled, can be set by the user with the form:

LOC e

where e must contain no undefined symbols when it is first encountered (during Pass 1).

The LOC instruction is most often used merely to define the memory address into which the first word assembled from a given program will ultimately be loaded. Otherwise, the first word will be located at zero (the LOCATION COUNTER is initialized to zero at the beginning of each pass).

EXAMPLES:

```
LOC  100
LOC A+25
```

The LOC instruction may be used to specify the beginning address of various segments of the same program. Also, a block of consecutive words may be reserved by

updating the LOCATION COUNTER relative to its current value. To reserve a block of 50 words and to label the first such word as AREA1, one would write

```
AREA1: LOC .+50
```

Note that the symbol AREA1 is assigned its value before the LOCATION COUNTER is updated.

4.4 PROGRAM TERMINATORS

The last statement of a source program must be

```
END
```

which causes the assembler to finalize all processing for the current pass and come to a halt.

If a source program consists of segments residing on different tapes, then each tape but the last should be terminated by the statement

```
EOT
```

which causes the assembler to pause for the insertion of the next tape into the reader.

CHAPTER FIVE

USAGE NOTES

This chapter describes conventions regarding subroutine linkage and presents further features of the assembler itself.

5.1 SUBROUTINE LINKAGE

The standard transfer of control to a subroutine in the GRI-909 is via a data test instruction. The JU (unconditional) or JC (conditional) jump instruction is used as appropriate. When any data test instruction results in a jump, the processor's sequence counter (SC) points to the second word of the jump instruction immediately before the jump takes place -- at this point the SC is transmitted to the trap (TRP), a hardware register associated with the data tester. Then the contents of the second word (or the incremented contents of the word it points to if deferred addressing is selected) is transmitted to the SC. The SC now points to the first (or entry) instruction of the subroutine called -- this instruction is executed next by the processor.

After any data testing jump is executed, the contents of TRP enables the return of control to the calling program if the jump was to a subroutine. Note that the address value in TRP is one less than that of next instruction in the calling program. If the subroutine called does not alter the contents of the trap register, either with a data test or a data transmission instruction, then the subroutine may return control by executing the instruction

```
RR   TRP,SC
```

Since the SC is automatically incremented <u>after</u> this instruction is executed, an absolute return of control to the proper location is performed.

If, on the other hand, the contents of TRP is likely to be affected by the subroutine itself, then the subroutine entry point instruction might be

```
SUB:  RMI  TRP,0
```

where the contents of TRP is stored into the second word of the entry instruction. The

subroutine may return control to the calling program via any one of the following instructions:

```
        JUD  SUB+1
        JCD  device, test, SUB+1
or      MR   SUB+1,P1,SC
```

Since the last instruction (MR) is a three-cycle instruction, the automatic incrementation of SC is completed before the instruction itself is executed - therefore, the instruction must increment the value being transmitted to SC.

A subroutine usually performs some operation or operations on one or more data items, called the arguments of the subroutine. Arguments are sometimes passed to subroutines by loading them into specific hardware registers before calling the subroutines. Also, arguments may be passed by following the subroutine call with a list of word values which define the arguments:

```
JU   SUB
WRD  V1
WRD  V2
WRD  V3
 .
 .
 .
WRD  Vn
```

where any Vi might be one of the following:

a) an address of data to be operated upon,
b) an actual data value to be operated upon,
c) an address to which return is made if errors are detected by the subroutine, or
d) an address into which results are to be stored.

If the subroutine entry instruction is

```
SUB:  RMI  TRP,0
```

then the first argument (V1) can be loaded into the AX register by

```
MRD  SUB+1,AX
```

The second and successive arguments can be fetched by executing similar instructions. Note that the word at SUB+1 is auto-incremented during each such deferred mode

instruction. When all the arguments have been picked up the word at SUB+1 contains one less than the normal return address -- the JUD SUB+1 (or its equivalent) is used for normal return of control to the calling program.

5.2 SYSTEM LINKAGE

When implementing and testing a large program which calls several user-generated subroutines and/or selected GRI utility routines, both assembly time and assembler symbol table space utilization may be minimized by assembling the subroutines together as a separate package. This package should contain only subroutines that have been checked out and it should be located (4.3) so as to reside in a memory area other than that occupied by the rest (main portion) of the program being developed. This process yields two object tapes to be loaded when running the program -- the main program and the subroutine package.

Once the subroutine package has been assembled, the entry point to each routine is at a known memory address. When re-assembling the main portion of the program, it is necessary to establish the linkage between it and the subroutines in the package. This is accomplished by preparing a series of parameter assignment statements (2.4) to be assembled with the main program. Given a set of subroutines Si starting at the corresponding locations Ni, the linkage to them is established by the statements

```
S1=N1
S2=N2
S3=N3
 .
 .
 .
```

When assembling from paper tape, these parameter assignment statements should be prepared on a separate source tape terminated with an EOT instruction (4.4) -- each time the main program is assembled it is only necessary to read these linkage statements during PASS 1 of the assembly process.

5.3 PSEUDO INSTRUCTIONS

In addition to statements containing standard predefined machine instruction codes, the assembler accepts statements of the form

$$[\text{symbol:}] \quad \left\{\begin{matrix}\text{constant}\\ \text{symbol}\end{matrix}\right\} [\text{e}] \quad [\text{comment}]$$

where the item in { } is called a pseudo instruction. The value of the constant or symbol is assembled into a single word and is displayed on the assembly listing in machine instruction format. The expression e, if present, is assembled into the next word and is displayed as data. The value of a symbolic pseudo instruction must be established via a parameter assignment statement (see 2.4).

The user may employ pseudo instructions to provide short and meaningful forms for machine instructions. For example, the assembly language instruction "FOM CLL" assembles as 02 0001 00 which has the octal value 004100. Redefining the standard symbol CLL with the statement "CLL=004100", the user may now code the clear link instruction by writing the pseudo instruction CLL. Such a redefinition must, of course, be included in each program that uses this symbol as a pseudo instruction. Another example is to replace the instruction "RR MSR,L1,0", which copies the bus overflow indicator into the link, by a symbol such as BVLNK whose value would be 037000.

The more common function of pseudo instructions is to enable the coding of commands that are arguments to interpretive subroutines. A call (JU) to an interpretive subroutine is followed by a sequence of commands that are arguments for the subroutine - the subroutine fetches and interprets each such command and performs the operation implied by the command. The GRI-909 Floating Point Interpreter ($SFI) maintains a software floating point accumulator. Floating point computations are invoked by commands to $SFI, where each command represents an operation to be performed on the software accumulator and calling program floating point data. For instance, to compute $Y=AX^2+BX+C$ in floating point, one would write the following assembly language instructions:

```
JU    $SFI
FLDA  A
FMPY  X
FADD  B
FMPY  X
FADD  C
FSTA  Y     .
FEXT
```

where the single-word pseudo instruction FEXT causes the subroutine to return control to the calling program. The other pseudo instructions each assemble into two words -- a command followed by the address of a floating point operand. For a complete

description of $SFI and its commands, see the GRI manual 74-44-001, "Floating Point Manual".

APPENDIX A

OPERATING INSTRUCTIONS

Passes 1,2 and 3 of the assembler perform user symbol definition, object code output and listing output respectively. After Pass 1, the assembler will continue to Pass 2 and then to Pass 3. Any time after Pass 1 has been completed, however, the assembler may be re-started and either Pass 2 or 3 selected.

I. Load the assembler with the Absolute Loader.

II. Transmit "0" to SC.

III. Set console switches as follows:

Bit 15 selects source input device
Bit 14 selects object output device
Bit 13 selects listing output device

UP = High-speed

DOWN = Low-speed (teletype)

Bits 1-0 select Pass

01 = Pass 1
10 = Pass 2
11 = Pass 3

} if Pass 1 previously completed.

IV. Ready source tape in reader (if TTI, set reader control to START).

V. Press START.

The assembler will halt after encountering an EOT mnemonic. Mount the next tape segment and press START.

The assembler will halt after encountering an END mnemonic. If another pass is either desired or necessary, remount the source tape (or the first segment thereof) and

a) press START to proceed to the next pass, or

b) select desired pass by starting at II, above.

At the beginning of pass 2 (before it is started), turn the punch ON if the object output is on TTO and turn it OFF after the pass is completed.

NOTES:

1) If bits 14 and 13 have different settings, then both the object code and the listing will be generated during Pass 2. The listing may be punched on the high-speed device and later printed off-line.

2) If the user wishes to type in instructions to see how various forms are assembled, procede as follows:
   a) Perform I and II, above.
   b) Perform III, selecting low-speed I/O (bits 15, 14, 13 of SWR down) and pass 3.
   c) Press START

Statements (each followed by a carriage-return) may now be typed on the TTY keyboard. The characters typed are not echoed on the teleprinter as they are struck. After a statement is terminated (carriage-return), the assembler responds with the corresponding listing output.

3) Patch tapes to correct logical errors in an assembled program may be created via the TTY keyboard as follows:
   a) Perform I and II, above.
   b) Perform III, selecting low-speed I/O (bits 15, 14, 13 of SWR down) and pass 2.
   c) Turn teletype punch "on".
   d) Press START - punch will generate leader.
   e) Type in patches, e.g.

```
LOC   100⏎   (⏎ denotes carriage return)
RS    AX,P1⏎
  :
etc.:
```

   f) No keyboard output will appear. No object punching will occur until another LOC statement or an END statement is typed in.
   g) Load object of original program.
   h) Load object tape created above.

Note: If high-speed equipment is available, do the same steps as above except: in b) leave bit 14 up, in c) turn high-speed punch "on", and f) will allow line typed in to be printed out on the keyboard in assembly listing format after the carriage return is typed in.

APPENDIX B

INSTRUCTION SUMMARY

MACHINE INSTRUCTIONS

The following symbols represent assembly language expressions having context-dependent meanings:

device - a source or destination device; SDA or DDA

pulse - a pulse output code; MOD

status - a status test code; MOD

test - a data test code; MOD (9-7)

path - a bus modifier path code; MOD (9-8)

location-a memory address or data value; full second word of memory reference instruction.

Function Generate

- general	FO *pulse, device*
- to machine	FOM *pulse*
- to interrupt control	FOI *pulse*
- to arithmetic operator	FOA *pulse*

Function Test

- general	SF *device, status*
- machine	SFM *status*
- arithmetic operator	SFA *status*

Data Test

- general	JC[D] *device, test, location*
- unconditional jump	JU[D] *location*

Data Transmit

- register to register	RR[C] *device* [,*path*], *device*
- zero to register	ZR[C] [*path*,] *device*
- register to self	RS[C] *device* [,*path*]
- register to memory	RM[I][D] *device* [,*path*], *location*
- zero to memory	ZM[I][D] [*path*,] *location*
- memory to register	MR[I][D] *location* [,*path*], *device*
- memory to self	MS[I][D] *location* [,*path*]

ASSEMBLER INSTRUCTIONS

e - denotes general assembly language expression

Data Definition

- transient parameter	*symbol* = e
- text	ASC $dc_1c_2c_3...c_nd$
- word values	WRD e [,e]...
- packed bytes	PKB e,e

Radix Selection

- octal	OCTAL
- decimal	DECIM[AL]

Set Location

- general	LOC e
- reserve n words	LOC . + n

Program Terminators

- end tape segment	EOT
- end program	END

APPENDIX C

STANDARD SYMBOL TABLE

The following are the pre-defined parameters that are part of the assembler's symbol table, to which user symbols are added.

INTENDED CATEGORY	SYMBOL	VALUE	MEANING
Device Addresses	ISR	4	Interrupt Status Register
	TRP	3	Trap Register
	SC	7	Sequence Counter
	SWR	10	Console Switch Register
	AX	11	Arithmetic Operator X-register
	AY	12	Arithmetic Operator Y-register
	AO	13	Arithmetic Operator
	MSR	17	Machine Status Register
	HSR	76	High-speed Reader
	HSP	76	High-speed Punch
	TTI	77	Teletype Input
	TTO	77	Teletype Output
Status Test Codes	AOV	2	Arithmetic Overflow
	SOV	4	Sum Overflow
	NOT	1	Negation of Test Results
	IRDY	10	Input-ready Flag
	ORDY	2	Output-ready Flag
	LNK	4	Bus Modifier Link
	BOV	2	Bus Overflow
	POK	10	Power OK
Transmission Path Codes	P1	1	Increment
	L1	2	Shift Left 1 Bit
	R1	3	Shift Right 1 Bit
Pulse Output Codes	CLL	1	Clear Link
	STL	2	Set Link
	CML	3	Complement Link
	HLT	4	Halt Machine

INTENDED CATEGORY	SYMBOL	VALUE	MEANING
Pulse Output Codes (continued)	ADD	0	Select AO "ADD"
	AND	4	Select AO "AND"
	OR	14	Select AO "OR"
	XOR	10	Select AO "XOR"
	STRT	1	General Start Pulse
	CLIF	10	Clear Input Flag
	CLOF	2	Clear Output Flag
	ICF	1	Interrupt Control OFF
	ICO	2	Interrupt Control ON
Data Test Codes	ETZ	2	Equal to Zero
	NEZ	3	Not Equal to Zero
	GTZ	7	Greater Than Zero
	GEZ	5	Greater Than or Equal to Zero
	LTZ	4	Less Than Zero
	LEZ	6	Less Than or Equal to Zero
Pseudo Codes	NOP	0	No Operation

SYMBOL TABLE CAPACITY

The B revision of %BAS has 43_{10} permanent symbols defined in it. The user symbol table may be built up to loc. 7607_8. Since symbols consist of three words per symbol, the table has additional capacity for 191_{10} user defined symbols. The number of symbols that make up the symbol table (including the permanent symbols) is stored in NSMAX, loc. 6311_8. For users with 8K of memory wishing to extend their symbol table capacity down to loc. 17607_8, for example, may do this by simply changing the contents of loc. 6311 to 3100_8. This would provide for up to 1557_{10} user defined symbols. In summary:

LOC.	CONTENTS	BOTTOM OF SYMBOL TABLE
6311_8	352_8	7607_8
6311_8	3100_8	17607_8

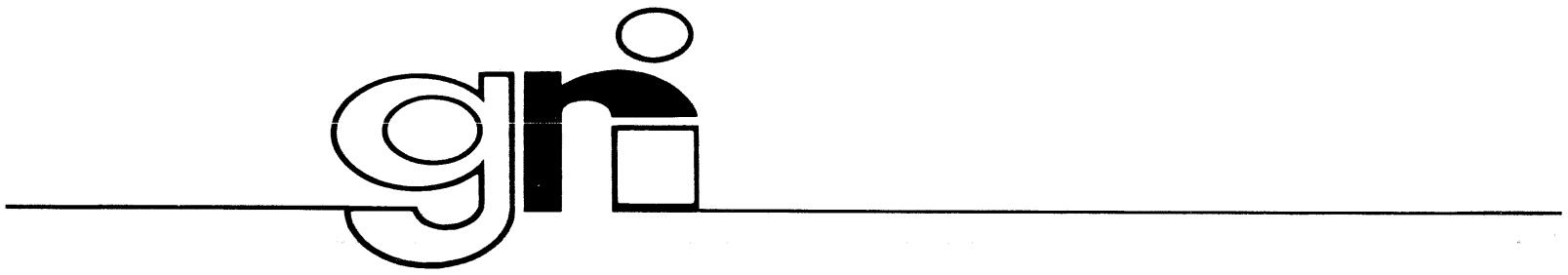
Any value may be calculated for this bottom location as follows:

$$NSMAX_8 = (LA_8 - 6312_8 + 1)/3_8 \quad \text{(NSMAX must be an integer)}$$

The number of user symbols (in octal) available is calculated as follows:

$$\#\ \text{user symbol}_8 = NSMAX_8 - 53_8$$

There were several minor bugs in the assembler, all of which have been taken care of in the B revision tape.



GRI Computer Corporation

320 NEEDHAM STREET, NEWTON, MASSACHUSETTS 02164

TEL: (617) 969-0800