



**relocatable
assembler
(% rasx)**



GRI Computer Corporation

320 NEEDHAM STREET, NEWTON, MASSACHUSETTS 02164

GRI-99
Relocatable Assembler
(%RASX)

GRI Computer Corporation, 320 Needham Street, Newton, Massachusetts 02164
Copyright © 1972 by GRI Computer Corporation

Issued: March, 1972

Supercedes: None

71-54-003-A
0372-200

TABLE OF CONTENTS

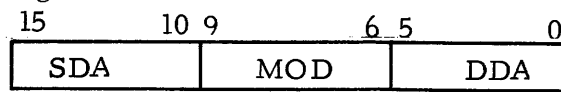
CHAPTER ONE - The Relocatable Assembler	1-1
1.1 Introduction	1-1
1.2 Assembler Output	1-4
CHAPTER TWO - Language Elements	2-1
2.1 Character Set	2-1
2.2 Symbols	2-3
2.3 Labels	2-5
2.4 Parameters	2-6
2.5 Constants	2-7
2.6 Expressions	2-7
2.7 Comments	2-10
2.8 Statements	2-10
CHAPTER THREE - Machine Instructions	3-1
3.1 Function generation	3-2
3.2 Function testing	3-3
3.3 Data testing	3-5
3.4 Data transmission	3-6
3.4.1 Non-memory transmission	3-7
3.4.2 Memory reference transmission	3-9
3.4.3 Indexing	3-10
CHAPTER FOUR - Assembler Instructions	4-1
4.1 Data definition	4-1
4.1.1 Text	4-2
4.1.2 Word values	4-2
4.1.3 Packed bytes	4-3
4.2 Radix	4-4
4.3 Set location	4-4
4.4 Program terminators	4-6
4.5 Listing control	4-6
4.6 Entry	4-7

TABLE OF CONTENTS (continued)

CHAPTER FIVE - Usage Notes	5-1
5.1 Subroutine linkage	5-1
5.2 System linkage	5-7
5.3 Pseudo instructions	5-8
APPENDIX A - Operating Instructions	A-1
APPENDIX B - Instruction Summary	B-1
APPENDIX C - Standard Symbol Table	C-1

C H A P T E R O N E
THE RELOCATABLE ASSEMBLER - %RASX

The GRI-99 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-99 consists of a sequence of binary coded machine instructions and data to be operated upon. GRI-99 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 Relocatable 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 relocatable assembler is an indispensable aid to the process of preparing binary object programs for the GRI-99 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.

The assembler, then, is simply a translator which translates an assembly language source program into its equivalent binary object form.

%RASX is always in one of two modes, relocatable mode or absolute mode. There are assembler instructions (see section 4.3) which the user can give to tell %RASX to switch from one mode to the other. When %RASX is in absolute mode, an internal variable called the ABSOLUTE LOCATION COUNTER is maintained. This internal variable continuously reflects the object program memory address for which source statements are being assembled in absolute mode. As each statement is read under absolute mode, the ABSOLUTE LOCATION COUNTER is updated by the number of machine words that will be occupied by the assembled statement. There is a similar internal variable, called the RELOCATABLE LOCATION COUNTER, which is maintained while the assembler

is in relocatable mode.

The user can label source statements with symbolic names. When such a label is encountered the symbol is defined by associating the current value and mode of the location counter with the symbol name. The location may then be referenced by its symbolic name rather than by its actual octal address. Such a symbolic reference can be made from other points in the same program or from points in another program. An undefined or external symbol is a reference in one program to a symbol which is defined in another program. Such cross references between programs are resolved by the loader at load time. %RASX has the very powerful feature of allowing the user to form arithmetic expressions involving external symbols. This is covered in section 2.6.

Since a symbol may be referenced in a program before it is defined, %RASX must read the source tape 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 on paper tape which can be loaded later by %RLHX. An optional third pass yields an assembly listing described in the next section.

Object tapes generated by %RASX are loaded via the relocatable loader %RLHX. When %RASX is in absolute mode, an initial address for the ABSOLUTE LOCATION COUNTER is specified by the user and the absolute object code which is generated will be loaded starting at this address. Under relocatable mode, the initial value of the RELOCATABLE LOCATION COUNTER is not specified by the user. Rather, the code is assembled relative to zero. Then at load time the user specifies the load address where the relocatable object code is to be stored.

When a label is encountered in relocatable mode it has an assembled value relative to zero. At load time the effective value of the label is its value relative to zero plus the load address. This relocatability feature means a relocatable object tape can be loaded anywhere in memory since the load address is

specified at load time rather than at assembly time.

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 values of user introduced symbols defined during PASS 1, generates the corresponding object program. The assembler punches the object program onto paper tape and the object program is subsequently loaded into the GRI-99 via the relocatable loader %RLHX.

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.

LINE #	LOCATION	RELOCATABILITY	DATA	LABEL	INSTRUCTION	OPERANDS	COMMENTS
*001					ENTRY	COMP,Y2	
002	00000	0	02 0000	13 COMP:	FOA	ADD	;ADD,SBTRCT SUBR.
003	00001	0	06 0010	11	MRI	X1,AX	;ARG. 1
	00002	1	000002				
004		1	000002	X1 = -1			
U 005	00003	0	06 0000	12	MR	X2,AY	;ARG. 2 IS EXTERNAL
	00004	0	000000				
U 006	00005	0	13 0000	06	RM	A0,Y1	;SUM IS EXTERNAL
	00006	0	000000				
007	00007	0	12 0110	12	RSC	AY,P1	;-ARG. 2
008	00010	0	13 0000	06	RM	A0,Y2	;DIFFERENCE
	00011	1	000013				
009	00012	0	03 0000	07 RTRN:	RR	TRP,SC	;RETURN FROM COMP
010	00013	0	000000	Y2:	WRD	0	
011		1	000014		END		

The * in column one is printed only on the first line of a block.

The relocatability attribute will always be zero, one or two indicating absolute, plus relocatable or minus relocatable respectively.

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 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 subfields 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.

For parameter assignment statements there is no address printed in the location field. However, the six digit value of the expression is printed starting in column 20.

PASS 3 of the assembler also generates a listing of user-introduced symbols and their assigned values. A single symbol, its relocatability attribute and its octal value, appear on each line. The error code U (shown in the following table) is printed before a symbol if appropriate. The code N is printed before the symbol in the symbol table if the symbol has been declared an entry point (see section 4.6).

A maximum of two error codes will be printed before the line number of any statement with errors. Since no more than two error codes can be printed, only the last two errors discovered are printed. The possible error codes and their meaning are:

- X External symbol in field where external symbol is not allowed (for example, in the MOD field of a data transfer instruction) or an external symbol is preceded by the operator ! or & %RASX assumes a value of zero and an absolute attribute for the external symbol.
- M Multiply defined label symbol. %RASX uses the value and attribute from the first definition.
- R Relocatability error: an expression did not resolve properly to absolute, plus relocatable or minus relocatable (see section 2. 6). The expression will not relocate properly at load time.
- S Syntax error: the statement is not formed according to the rules defined in this manual. The assembler generates two zero words as object output.
- V Symbol table overflow: the program contains more symbols than will fit in the symbol table. All symbols defined after the symbol table has become full will have a value of zero and an absolute attribute.
- U Undefined symbol (also called external symbol): treated as zero with an absolute attribute. References to external symbols are resolved at load time (see section 5. 2).

- < Too few expressions in operands field.
- > Too many expressions in operands field.
- W Warning, two or more adjacent operators in an expression or operand missing in expression. %RASX treats A+-B as A+B and WRD A, , B as WRD A, \emptyset , B.
- D Decimal digit 8 or 9 is in octal field: a numeric constant contains an 8 or 9 and the assembler's radix is set to octal.
- T Truncation error. Too many expressions (more than 19) in operands field: or ASC statement terminated by carriage return. (This can mean either than the ASC statement was legitimately terminated by a carriage return or that it contained more than 75₁₀ characters and so was truncated.)
- N Symbol has been declared as an entry point. Note that this is not an error, it simply flags that symbol as an entry point, and appears only in the symbol table printout preceding the listing.
- E Entry point error: the ENTRY statement is out of place or improperly formed.
- I Indexing error; # appears in field where indexing is not allowed.

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-99 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:

<u>Character</u>	<u>External</u>	<u>Internal</u>
Alphabetics	A through Z	301 through 322
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:

<u>Character</u>	<u>External</u>	<u>Internal</u>	<u>Function</u>
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.

<u>Character</u>	<u>External</u>	<u>Internal</u>	<u>Function</u>
Back-arrow	←	337	Causes the first previous input character not a back-arrow to be ignored by the assembler.
Number sign	#	243	Denotes indexing
Block mark		375 or 233	Separates block 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.

User defined symbols have an assembled value and an attribute. The attribute of a symbol determines how the value of the symbol is treated at load time. A symbol has one of three attributes: absolute, plus relocatable, or minus relocatable. If the symbol is absolute the final value of the symbol is **simply the value assigned at assembly time.** If a symbol is plus (minus) relocatable **the final value of the symbol is its assembled value plus (minus) the load address at load time.** Any indexing (specified by #) will cause bit 15 of the field to be set at load time after all relocation arithmetic has been done.

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
```

The following symbols are invalid for the reasons given. Invalid symbols may cause a syntax error and generate two zero words for object output.

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 definition results in a symbol whose attribute is either absolute or plus relocatable. If %RASX is in absolute mode the current value of the ABSOLUTE LOCATION COUNTER is assigned to the symbol and its attribute is absolute. If %RASX is in relocatable mode the current value of the RELOCATABLE LOCATION COUNTER is assigned to the symbol and its attribute is plus relocatable.

A label is used to symbolically reference a specific instruction or data word from other points in the same program or from points in another program. Therefore, if a user attempts to associate a label with two or more different memory addresses the error code M, indicating a multiply defined label, is printed in the assembly listing. It is also possible to discover an M error at load time. This is discussed in detail in section 4.6.

EXAMPLES:

TYPE: RR AX, TTO

TABLE: WRD - 114, - 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 and attribute of the parameter will be the assembled value and attribute 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 and attribute to be assigned to the parameter symbol. An assembly parameter, unlike a label symbol, may be redefined within the same program.

To ensure that the proper value of a parameter is used the parameter must be completely defined before it is referenced in PASS 2. In the following example the first reference to A will be wrong on PASS 2 since the reference to A occurs before A is completely defined. The second reference to A will be correct.

```
MRI          A, AY          ; 1st REFERENCE TO A
```

```
A = B + 5
```

```
B = 6
```

```
MRI          A, AX          ; 2nd REFERENCE TO A
```

A is completely defined when the statement A = B + 5 is encountered on PASS 2.

It is possible to define parameters in such a way that they are not completely defined until PASS 3. For example:

$$X = Y + 5$$

$$Y = Z - 2$$

$$Z = 24$$

The values assigned to the symbols in this example when their defining statements are encountered will be as follows:

	X	Y	Z
PASS 1	5	-2	24
PASS 2	3	22	24
PASS 3	27	22	24

The value of X is different on all three passes: X is not completely defined until PASS 3.

2.5 CONSTANTS

A simple constant is represented by one or more successive numeric characters. The character string is converted ignoring overflow 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.

A simple constant is considered to have an absolute attribute.

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 operations, where the first such operand may be preceded by an arithmetic operator (leading sign). Any given

operand may be one of the following:

<u>Operand</u>	<u>Attribute</u>
Label	absolute, plus relocatable
Parameter	absolute, plus relocatable, minus relocatable
Constant	absolute
· (representing assembler's current location counter)	absolute, plus relocatable

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

$$\left[\left\{ \begin{array}{c} + \\ - \end{array} \right\} \right] \left\{ \begin{array}{c} \text{Label} \\ \text{Parameter} \\ \text{Constant} \end{array} \right\} \left[\left\{ \begin{array}{c} + \\ - \\ \& \\ ! \\ \text{Space} \end{array} \right\} \left\{ \begin{array}{c} \text{Label} \\ \text{Parameter} \\ \text{Constant} \end{array} \right\} \right] \dots$$

A W error for warning is printed if an expression contains two or more adjacent operators. Only the first operator is used; for example A+!B is treated as A + B. W error is also printed if an operand is missing. For example, WRD,, is treated as WRD ϕ , ϕ , ϕ .

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

EXAMPLES:

15

-237

A + B

. + B - 3

VAL!VAL2&VAL3

The attribute of an expression is calculated as follows. Assign absolute symbols a value of \emptyset , plus relocatable symbols a value of +1 and minus relocatable symbols a value of -1. Evaluate the expression with these values. If the result is \emptyset the expression is absolute, if +1 the expression is plus relocatable, and if -1 the expression is minus relocatable. Any other result causes an R error to be printed on the assembly listing and the result will not relocate properly at load time. For example the attribute of the following expression (where TABLE and START are plus relocatable) is absolute.

$$X = \text{TABLE} + 5 - \text{START}$$

$$(+1) + (\emptyset) - (+1) = \emptyset$$

However, the following expression results in an R error:

$$X = -\text{TABLE} - \text{START}$$

$$-(+1) - (+1) = -2$$

Also, if the AND or OR operators are used, the attribute of the expression up to that operator and the attribute of the symbol to the right of that operator must be absolute or an R error is generated. For example an R error is caused by

$$\text{RM AX,5 ! START}$$

but the following expression is okay.

$$\text{MRI TABLE - START ! 5, AY}$$

%RASX allows undefined or external symbols in an expression. They are treated as absolute with a value of zero for purposes of evaluating the attribute and value of the expression. However, at load time the value of the expression will include the values of the undefined symbols if they are declared as entry points in another program.

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 Three, Four and Five 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 for 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.

CHAPTER THREE

MACHINE INSTRUCTIONS

Although all basic GRI-99 machine instructions have the same format -- SDA MOD DDA - (See Chapter One), 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 Test -- 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

02	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, O

FO AND, AO

The assembler provides mnemonics which imply a specific destination.

These are of the form:

- machine (control logic)

FOM e

02	e	00
----	---	----

- interrupt control

FOI e

02	e	04
----	---	----

- arithmetic operator

FOA e

02	e	13
----	---	----

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 e₁, e₂

e ₁	e ₂	O 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 TTY, ORDY

SF HSP, NOT ORDY

SF O, BOV

SF AO, AOV

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

- machine

SFM e

00	e	02
----	---	----

- arithmetic oj

SFA e

13	e	02
----	---	----

EXAMPLES:

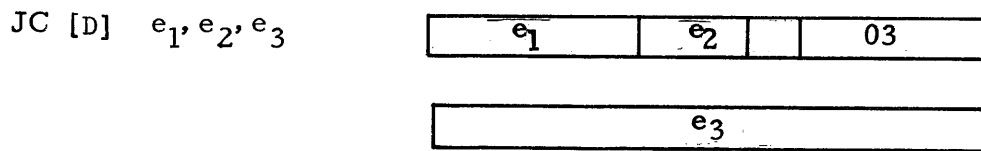
SFM BOV ; BUS OVERFLOW SET?

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:



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 to e_3 on the "OR" of the truth of the selected test conditions.

1- jump to e_3 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:

<u>CODE</u>	<u>VALUE</u>	<u>CONDITION</u>
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:

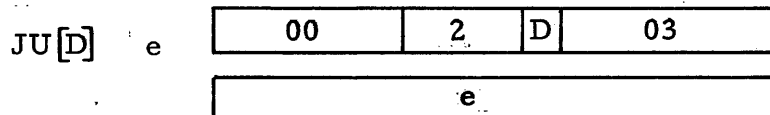
```
JC  AX,GEZ, LOOP+5
```

```
JC  TTI,ETZ, AGAIN
```

```
JCD AO, LTZ, SUB+1
```

```
JC  O, 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:



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:

L1 - Shift left one bit

R1 - Shift right one bit

P1 - Increment (add one)

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

e_1	e_2	C	O	e_3
-------	-------	---	---	-------

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

EXAMPLES:

RR O, AX ; CLEAR AX
 RRC AO, P1, AX ; 2's COMP OF AO TO AX
 RR TTL, 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

e_1	e_2	C	O	e_1
-------	-------	---	---	-------

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 both 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

00	e_1	C	O	e_2
----	-------	---	---	-------

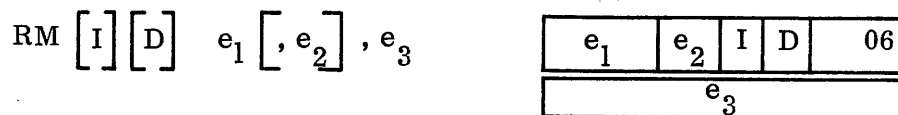
EXAMPLES:

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:



EXAMPLES:

RM AX, SAVE1

RM AO, P1, Z1+5 ; STORE OUTPUT+1

RMI TRP, O ; STORE TRAP IMMEDIATE

RMD AX, ADDR

The clearing of memory locations is facilitated by the form:

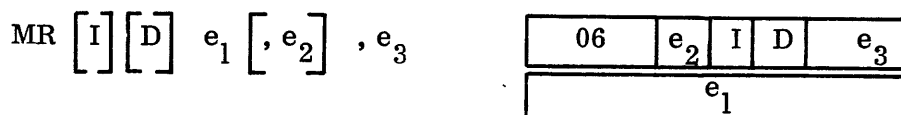


EXAMPLES:

ZM COUNT ; CLEAR COUNTER

ZM P1, SW1 ; SET SWITCH

Registers may be loaded from memory using the general form:



EXAMPLES:

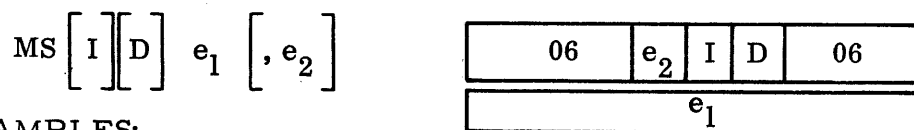
MR SAVE1, AX ; RESTORE AX

MRI Z15, TTO ; TYPE CARRIAGE-RETURN

MRD A, L1, AX

MRI -1, TRP ; TRAP=-1

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



EXAMPLES:

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

3.4.3 INDEXING

Address words, e.g., the second word of a two word instruction, may specify that the address word of the instruction is to be indexed to establish the effective address. This is signaled by bit 15 of an address word being set to a one. The assembler is directed to set bit 15 of a word of object by preceding the expression which defines the word with a #. Thus

```
MR   #LABL, AX
```

```
or   JC   AX, ETZ, #SUBR+3
```

signals the assembler to generate the index bit of the word into which the value of the expression will be stored. The assembler does not actually set this bit on the object tape - because the value of the expression may need relocation which must occur before the bit is set. Instead, separate information is output on the object tape to inform the loader that bit 15 is to be set after relocation (if any) has occurred. The listing, however, will show bit 15 set to aid in debugging.

The index symbol must occur at the beginning of the expression in which it occurs, otherwise a syntax error (S) will occur. If indexing is specified for a field which does not correspond to a 16 bit (full word) quantity of object code, an indexing error (I) will occur. Thus, valid examples are

WRD #LABEL+5, XYZ, 1, #3
 MRID #Ø, AX
 JU #5
 RM AX, #EXTRN+6
 JC AX, ETZ, #RETRN

Examples causing syntax errors are

WRD LABEL+#5
 JU EXTRN#+6

Examples causing index errors are

PARAM=	#e	(parameter definitions do not generate 16 bit object)
JC	#AX, ETZ, LABEL	(#AX occurs in a field corresponding to a <u>6</u> bit register number)
MR	LABEL, #P1, TRP	(#P1 specifies a <u>4</u> bit mod field)

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

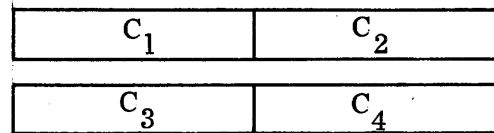
4.1 DATA DEFINITION

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 $dc_1c_2c_3c_4\cdots c_n d$



⋮

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 (or a carriage return), and the c_i are text characters.

The text delimited by the d 's is assembled into consecutive words, (considered absolute) **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. If the text contains no characters, two words of zero are generated as object output and an S (syntax) error occurs. 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. (The delimiter may be any character except back arrow, rubout, carriage return or colon.)

```

MSC:    ASC    /MOUNT NEXT TAPE/
        ASC    'A/B=LIM1'
ALRMI:  ASC    .ALARM '1'.

```

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

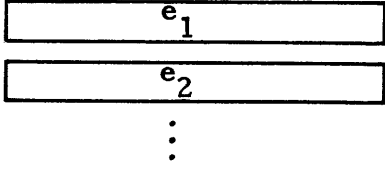
4.1.2 WORD VALUES

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

```

WRD  e1 [ e2 ] ...

```



The values of one or more expressions are assembled into the corresponding number of consecutive words. The attribute associated with each word will be the attribute of the corresponding expression.

EXAMPLES:

```

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

```

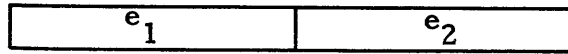
A label associated with a WRD instruction may be used to reference the first word assembled therefrom.

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

PKB CR, LF

If e_1 or e_2 contain any external symbols, an X error is printed and the external symbol is treated as zero with an absolute attribute.

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 each 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

An assembler's location counter continuously reflects the memory address into which a source statement is being assembled. As stated in the introduction %RASX maintains two location counters. The ABSOLUTE LOCATION COUNTER is used to associate memory locations with source statements assembled in absolute mode and is therefore updated only when the assembler is in absolute mode. The RELOCATABLE LOCATION COUNTER associates memory locations with source statements assembled in relocatable mode and as such is only updated when the assembler is in relocatable mode.

At the beginning of each pass %RASX assumes relocatable mode and sets the value of the RELOCATABLE LOCATION COUNTER to zero. As each source statement is translated to object code during relocatable mode, the RELOCATABLE LOCATION COUNTER is updated by the number of words the source statement requires. At any time the user can switch to absolute mode by writing the instruction:

LOC e

where the expression e contains no undefined symbols when first encountered and e has an absolute attribute. If e has a relocatable attribute the assembler merely resets the RELOCATABLE LOCATION COUNTER to the value of e and continues in relocatable mode.

When the instruction LOC e is encountered and e has an absolute attribute, %RASX enters absolute mode and sets the ABSOLUTE LOCATION COUNTER to the value of e. As each source statement is assembled during absolute mode the ABSOLUTE LOCATION COUNTER is updated by the number of words the source statement requires.

The user can change from absolute mode to relocatable mode by writing the statement

REL

when %RASX encounters a REL statement it merely switches to relocatable mode and uses the latest value of RELOCATABLE LOCATION COUNTER.

Stated simply, the REL statement forces %RASX to relocatable mode. The LOC e statement sets %RASX to relocatable mode if e is relocatable and to absolute mode if e is absolute.

In addition to setting the mode of the assembler, the LOC e statement can be used to reserve a block of consecutive words by updating the location counter relative to its current value. For example to reserve a block of 50 words and to label the first such block AREAL, one would write

```
AREAL: LOC .+50
```

Note that the symbol AREAL 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.

4.5 LISTING CONTROL

There are three instructions to modify the assembly listing output. The instruction EJECT causes the assembler to go to the top of a page to continue listing. The EJECT instruction itself will be listed at the top of the page.

The statement NLIST causes the assembler to stop listing (but not to stop assembling) until a LIST or END command is encountered. The NLIST statement itself is printed. The assembler assumes LIST at the beginning of each pass.

4.6 ENTRY

The format for the ENTRY statement is:

```
ENTRY  s1 [, s2] .. [; comment]
```

where s1, s2... are symbol names. A symbol name should appear in an ENTRY statement in the program which defines the symbol if that symbol name is to be referenced from another program. When %RASX punches an object tape it also punches a table of all undefined symbols and a table of all symbols which have been declared as entry points (i. e. which appear in an ENTRY statement). At load time, the entry points defined in one program are used to resolve the undefined symbols in other programs. For example, if the following two programs were assembled and then loaded,

```

                                ; PROGRAM ONE
                                ENTRY A
A:                                WRD 15
                                END

                                ; PROGRAM TWO
                                MR    A, AX
                                END
```

the undefined symbol A in program TWO would be resolved by the loader to the value of A defined in program ONE since A is declared in program ONE to be

an entry point. If A had not been declared as an entry point in program ONE the loader would have been unable to resolve the reference to A in program TWO. The reference to A in program TWO may be preceded by # if the index bit is to be set when A is resolved.

The possibility exists of a symbol becoming multiply defined (M error) at load time (section 2.3 discusses multiple definition during assembly). If the same symbol name is declared as an entry point in two different programs and those two programs are loaded together the symbol will be multiply defined at load time. As with multiple definitions at assembly time, the first value of the symbol is used.

If there are any ENTRY statements in a program they must appear as the first statements of that program. An exception to the rule is that an ENTRY statement may be preceded by lines consisting of comments only.

An E error will be printed if the ENTRY statement is improperly formed or out of place.

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-99 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 TRP 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 after 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 incrementing 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   ARG1
WRD   ARG2
WRD   ARG3
      .
      .
      .
WRD   ARGn

```

where any ARG2 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 (ARG1) 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 of control to the calling program.

The index register may also be used to advantage in these situations. If the TRP register is going to be altered by the subroutine, instead of saying RMI TRP, 0 at the beginning one might say RR TRP, XR in which case (if the XR is not destroyed by the subroutine) a return can be effected by

```
RR XR, SC
```

Also, if arguments have been passed in the call, the subroutine can begin by saying RR TRP, XR and picking up arguments may proceed by

```
MR #1, AX          ; arg 1 to AX
```

```
.  
.
.
```

```
MR #2, AX          ; arg 2 to AX
```

and at the end, if n arguments are in the call, and the XR has not been destroyed, the return may be effected by

```
JU #n+1
```

or

```
MRI #n, SC          (contents of XR+n go to the SC, and
                    since MRI is a 2 cycle instruction,
                    the SC is bumped by one after
                    execution thereby effecting return
                    to the proper place)
```

This method of picking up arguments has the disadvantage that it destroys the XR, but there are many advantages -- every instruction involved is 1 cycle shorter, the arguments need not be picked up sequentially, and all of the arguments do not have to be picked up in order to effect a proper return; eg. compare

JU SUBR

WRD ARG1

WRD ARG2

.

.

.

WRD ARGn

Method 1

SUBR: RMI TRP, ϕ

MRD SUBR+1, AX ; ARG1 to AX

.

; CAN'T GET ARG 3 OR RETURN

MRD SUB+1, AX ; ARG2 to AX

.

.

.

MRD SUB+1, AY ; ARGn to AY (MUST BE

JUD SUB+1 ; DONE BEFORE RETURN)

Method 2

SURB: RR TRP, XR

MR #1, AX ; 1st ARG

.

.

.

MR #3, TRP ; 3rd ARG

MR #2, AY ; THEN 2rd ARG

.

.

; DO NOT HAVE TO GET

JC AO, ETZ, #n+1 ; ALL N ARGS BEFORE RETURN

```

      .
      .
      .
MR     #n, AX
etc.
JU     #n+1           ; ANOTHER RETURN

```

Note that RMI TRP, \emptyset (method 1) is 2 cycles, RR TRP, XR (method 2) is 1 cycle, MRD instructions (method 1) are 4 cycles, MR instructions (method 2) are 3 cycles and JUD (method 1) is 3 cycles whereas JU (method 2) is 2 cycles for the return.

The index register may be used as both a loop counter and as an address modifier at the same time as follows: suppose you wish to zero out a table which is 100_8 words long. The following will accomplish this:

```

MRI   -100, XR
ZM    #TABLE+100
RS    XR, P1
JN    .-3
      .
      .
      .

```

The first time through the loop the ZM instruction will zero $TABLE + 100 + XR = TABLE + 100 - 100 = TABLE$; the second time through the XR will be -77 since the RS XR, P1 added one to it - hence the ZM instruction will zero $TABLE + 100 + XR = TABLE + 100 - 77 = TABLE+1$; and so forth. The last time through the loop XR will equal -1 and the ZM instruction will zero $TABLE + 100 - 1 = TABLE + 77$ (the last word), then the RS XR, P1 bumps XR to zero which sets BOV so that the JN

instruction fails and the loop is done.

5.2 SYSTEM LINKAGE

Programs assembled by %RASX are linked together at load time. As discussed in section 4.6, when a symbol is defined in one program and is declared in that program to be an entry point, then that symbol can then be referenced by other programs. The references to the symbol from other programs will be undefined (external) at assembly time but will be resolved to the proper value at load time.

For example if TABA and LEN were undefined at assembly time the expression in the following statement would assemble to a value of zero and an absolute attribute.

```
LOCAT:  MRI  TABA+LEN, TRP
```

However, if the object tape was loaded with a tape which had declared TABA and LEN as entry points and had defined TABA as 10 and LEN as 20 (both absolute) then a 30 would be loaded into LOCAT+1. The expression is resolved to the proper value at load time.

If the defining program had defined TABA as plus relocatable with a value of 32 and LEN as absolute with a value of 24 and the defining program were loaded starting at location 2000 then TABA would have an effective value of $2000+32=2032$ (see section 2.2). Therefore, $2032+24=2056$ would be loaded into LOCAT+1. If, further, the statement had been LOCAT: MRI #TABA+LEN, TRP then $(2032+24)$ or'd with bit 15 = 102056 would be loaded into LOCAT+1.

5.3 PSEUDO INSTRUCTIONS

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

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

where the item in $\left\{ \begin{array}{l} \text{constant} \\ \text{symbol} \end{array} \right\}$ 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. Refining 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, Ll, 0", which copies the bus overflow indicator into the link, by a symbol such as BVLNK whose value would be 037000.

%RASX has defined the following pseudo instructions in the resident symbol table:

JO	Jump on BOV, equivalent to JC MSR, LTZ, address
JN	Jump on NOT BOV, equivalent to JC MSR, GEZ, address
JOD	Jump deferred on BOV
JND	Jump deferred on NOT BOV
CLL	Clear link, equivalent to FOM CLL
STL	Set link, equivalent to FOM STL
CML	Complement link, equivalent to FOM CML
HLT	Halt, equivalent to FOM HLT
ADD	Set AO to ADD, equivalent to FOA ADD
AND	Set AO to AND, equivalent to FOA AND
OR	Set AO to OR, equivalent to FOA OR
XOR	Set AO to XOR, equivalent to FOA XOR
ICO	Set interrupt control on, equivalent to FOI ICO
ICF	Set interrupt control off, equivalent to FOI ICF
CLB	Clear BOV, equivalent to ZR Pl, \emptyset
STB	Set BOV, equivalent to ZRC Pl, \emptyset
BTL	Copy BOV to LNK, equivalent to RR MSR, L1, \emptyset
ATL	Copy AOV to LNK, equivalent to RR MSR, R1, \emptyset
ZMS	Set MSR to \emptyset , equivalent to ZR MSR
SKP	Skip two words, equivalent to SFM NOT
NOP	No operation, equivalent to RR 0, 0

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-99 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, refer to GRI manual 74-44-001, "Floating Point Interpretive Language."

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. Once Pass 1 has been completed, however, the assembler may be re-started and either Pass 2 or 3 selected. Once Pass 2 has been run it should not be run again without going through Pass 1 first.

- 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) Since input is interrupt driven, the user must be sure to press START at all times rather than CONTINUE. Also he must be sure that the I/O card for the input device is plugged into the back of the computer in such a way that the PINL and POUTL lines can reach the card (see GRI-99 Systems Reference Manual). A sure way to accomplish this is to plug the card into the left-most slot in the back I/O bus.

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]

Pseudo Codes

- jump on BOV	JO [D]	location
- jump on NOT BOV	JN [D]	location
- clear link	CLL	
- set link	STL	
- complement link	CML	
- halt	HLT	
- add	ADD	
- and	AND	
- or	OR	
- exclusive or	XOR	
- interrupt on	ICO	
- interrupt off	ICF	
- clear BOV	CLB	
- set BOV	STB	
- copy BOV to LNK	BTL	
- copy AOV to LNK	ATL	

- | | |
|------------------|-----|
| - zero MSR | ZMS |
| - skip two words | SKP |
| - no operation | NOP |

ASSEMBLER INSTRUCTIONS

e - denotes general assembly language expression

Data Definition

- | | |
|-----------------------|---|
| - transient parameter | Symbol = e |
| - text | ASC dc ₁ c ₂ c ₃ ...c _n d |
| - 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 |
| - enter relocatable mode | REL |

Program Terminators

- | | |
|--------------------|-----|
| - end tape segment | EOT |
| - end program | END |

Listing Control

- | | |
|-----------------------|-------|
| - list at top of page | EJECT |
| - resume listing | LIST |
| - inhibit listing | NLIST |

Entry

- declare symbol as
entry point

ENTRY SYMBOL [SYMBOL] ...

A P P E N D I X C
S T A N D A R D S Y M B O L T A B L E

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

<u>INTENDED CATEGORY</u>	<u>SYMBOL</u>	<u>VALUE</u>	<u>MEANING</u>
Device Addresses	ISR	4	Interrupt Status Register
	TRP	23	Trap Register
	XR	22	Index 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

<u>Intended Category</u>	<u>Symbol</u>	<u>Value</u>	<u>Meaning</u>
	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
	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



 **GRI Computer Corporation**

320 NEEDHAM STREET, NEWTON, MASSACHUSETTS 02164

TEL: (617) 969-0800