MODEL 990 COMPUTER

ASSEMBLY LANGUAGE PROGRAMMER'S GUIDE

MANUAL NO. 943441-9701
ORIGINAL ISSUE 1 JUNE 1974

TEXAS INSTRUMENTS
INCORPORATED

LIST OF EFFECTIVE PAGES

INSERT LATEST CHANGED PAGES DESTROY SUPERSEDED PAGES

Note: The portion of the text affected by the changes is indicated by a vertical bar in the outer margins of the page.

Model 990 Computer Assembly Language Programmer's Guide (943441-9701)

Original Issue . . . . . 1 June 1974

Total number of pages in this publication is 94 consisting of the following:

PAGE NO.	CHANGE NO.	PAGE NO.	CHANGE NO.	PAGE NO.	CHANGE NO.
Cover.	0	8-1 - 8-3	0		
Eff. pages.	0	8-4 Blank	0		
iii - v.	0	9-1 - 9-2	0		
vi Blank	0	10-1 - 10-11 . . .	0		
vii	0	10-12 Blank. . . .	0		
viii Blank	0	App A Divider . .	0		
1-1	0	A-1 - A-2	0		
1-2 Blank	0	App B Divider . .	0		
2-1 - 2-7	0	B-1 - B-12	0		
2-8 Blank	0	App C Divider . .	0		
3-1 - 3-4	0	C-1 - C-12	0		
4-1 - 4-2	0	App D Divider . .	0		
5-1 - 4-9	0	D-1 - D-2	0		
5-10 Blank	0	User's Resp . . .	0		
6-1 - 6-6	0	Bus. Reply	0		
7-1 - 7-2	0	Cover Blank . . .	0		
		Back Cover	0		

TABLE OF CONTENTS

TABLE OF CONTENTS (Continued)

TABLE OF CONTENTS (Continued)

LIST OF ILLUSTRATIONS

LIST OF TABLES

SECTION I

GENERAL INFORMATION

1.1 SCOPE OF MANUAL

This manual contains detailed information about the Model 990 Computer Assembly Language, including source statement formats and elements, addressing modes, assembler directives and pseudo-instructions, and source statement formats for machine instructions. The manual also describes the assembler output, both the listing and object code. The appendices contain the character set, a sample program, and instruction and directive tables.

1.2 REFERENCES

The machine instructions of the Model 990 Computer are described in the Model 990 Computer Reference Manual. Input/Output subroutines and techniques are described in the Input/Output Assembly Language Users Guide.

1.3 MODEL 990 ASSEMBLER

The Model 990 Assembler is a one-pass assembler that assembles object code for the Model 990 Computer from assembly language source statements, and is implemented on the Model 990 Computer.

The Model 990 Assembler can assemble both absolute and relocatable object code in the same assembly. The object code is formatted to allow transmission over telephone lines, and to allow correction of object code on the object medium with an off-line device. The assembler processes external references and definitions, which permits jobs to be assembled as separate programs and linked together by the linking loader.

SECTION II

LANGUAGE REQUIREMENTS

2.1 SOURCE STATEMENT FORMAT

An assembly language source program consists of source statements which may contain assembler directives, machine instructions, pseudo-instructions, or comments. Each source statement is a source record as defined for the source medium. With the exception of comment statements, each statement may have as many as four fields: the label field, the operator field, the operand field, and the comment field. The fields are separated by one or more blanks; and no field, with the exception of the comment field, may contain embedded blanks. A tab character (CTRL I) may be used in place of a blank to separate fields on the ASR733 and the ASR33. Two acceptable formats of source statements are shown in figure 2-1. The first four lines show the fields aligned on arbitrarily chosen character positions to produce aligned fields in the source listing. The next four lines show the fields separated by tab characters.

Comment statements consist of a single field starting with an asterisk (*) in the first character position followed by any ASCII character including a blank in each succeeding character position. Comment statements are listed in the source portion of the assembly listing and have no other effect on the assembly.

The maximum length of source records is 60 characters. However, only the first 52 characters will be printed on the ASR733 or the ASR33. The end-of-record for the source medium is placed following the last field used.

2.1.1 CHARACTER SET

The Model 990 Assembler recognizes ASCII characters as follows:

- The alphabet (capital letters only) and space character
- The numerals
- Twenty-two special characters
- Five undefined characters
- The null character
- The tab character

Appendix A contains tables that list all 66 characters and show the ASCII and Hollerith codes for each.

2.1.2 LABEL FIELD

The label field begins in character position one of the source record and extends to the first blank. The label field contains a symbol (paragraph 2.4)

```
* CONVENTIONAL SOURCE STATEMENT FORMAT
START   LI    3,>25         LOAD W R 3
        A     5,3           ADD W R 5
        RT                  RETURN TO CALLING PROGRAM
* PACKED SOURCE STATEMENT FORMAT USING TABS
START^LI^3,>25^LOAD W R 3
^A^5,3^ADD W R 5
^RT^RETURN TO CALLING PROGRAM
```

(A)128440

Figure 2-1. Source Statement Formats

supplied by the programmer. A label is optional for machine instructions and pseudo-instructions, and for many assembler directives. When the label is omitted, the first character position must contain a blank. A source statement consisting of a label field only is a valid statement that has the effect of an EQU directive (paragraph 5.3.4) with the same label and with a dollar sign ($) in the operand field (paragraph 2.4.1).

CAUTION

When the location counter contains an odd location, and a source statement consisting only of a label is followed by a machine instruction or a DATA directive, the machine instruction or data word does not have the same location as the label.

2.1.3 OPERATOR FIELD

The operator field begins following the blank that terminates the label field, or in the first non-blank character position after the first character position when the label is omitted. The operator field is terminated by one or more blanks, and may not extend past character position 60 of the source record. The operator field contains a symbol that defines the operation, which may be a machine instruction, a pseudo-instruction, or an assembler directive. The operator field may contain a user-defined extended operation symbol (paragraph 5.5.1).

2.1.4 OPERAND FIELD

The operand field begins following the blank that terminates the operator field, and may not extend past character position 60 of the source record. The operand field may contain one or more expressions, terms, or constants, according to the requirements of the operation specified in the operator field. The operand field is terminated by one or more blanks.

2.1.5 COMMENT FIELD

The comment field begins following the blank that terminates the operand field, and may extend to the end of the source record if required. The comment field may contain any ASCII character, including blank. The contents of the comment field are listed in the source portion of the assembly listing and have no other effect on the assembly.

2.2 EXPRESSIONS

Expressions are used in the operand fields of assembler directives and machine instructions.

2.2.1 DEFINITION

An expression is a constant or symbol, or a series of constants, a series of symbols, or a series of constants and symbols separated by arithmetic operators. Each constant or symbol may be preceded by a minus sign (unary minus). The expression may contain no embedded blanks. The symbols may not be symbols that are defined as extended operations (paragraph 5.5.1). Symbols that are defined as external references (paragraph 5.4.2) may not be operands of arithmetic operations. Only one symbol in an expression may be subsequently defined in the program, but that symbol must not be part of an operand in a multiplication or division operation within the expression. An expression that contains a relocatable symbol or constant immediately following a multiplication or division operator is an illegal expression. Also, when the result of evaluating an expression up to a multiplication or division operator is relocatable, the expression is illegal. An expression in which the number of relocatable symbols or constants added to the expression minus the number of relocatable symbols or constants subtracted from the expression is not equal to zero or one is an illegal expression. Refer to paragraph 4.2 for definition of relocatability.

The following are examples of valid expressions:

```
BLUE+1

GREEN-4

2*16+RED

440/2-RED
```

2.2.2 WELL-DEFINED EXPRESSIONS

Some assembler directives require well-defined expressions in the operand fields. A well-defined expression must not contain any symbols or assembly-time constants that are not previously defined. No character constant may be placed in a well-defined expression. The evaluation of the entire expression must be absolute.

2.2.3 ARITHMETIC OPERATORS AND ORDER OF EVALUATION

The arithmetic operators in expressions are as follows:

- + for addition
- - for subtraction
- * for multiplication
- / for division

In evaluating an expression, the assembler first negates any constant or symbol preceded by a unary minus, then performs the arithmetic operations from left to right. The assembler does not assign precedence to any operation other than unary minus.

For example, the expression 4+5*2 would be evaluated 18, not 14. Also, the expression 7+1/2 would be evaluated 4, not 7.

2.3 CONSTANTS

Constants are used in expressions. The assembler recognizes four types of constants: decimal integer constants, hexadecimal integer constants, character constants, and assembly-time constants.

2.3.1 DECIMAL INTEGER CONSTANTS

A decimal integer constant is written as a string of numerals. When a decimal integer constant represents data, the range of values is -32,768 to +65,535. Positive decimal integer constants greater than 32,767 are considered negative when used as operands of addition and subtraction instructions.

The following are valid decimal constants:

```
 1000
-32768
  25
```

2.3.2 HEXADECIMAL INTEGER CONSTANTS

A hexadecimal integer constant is written as a string of up to four hexadecimal numerals preceded by a greater than (>) character. Hexadecimal numerals include the decimal values 0 through 9 and the letters A through F.

The following are valid hexadecimal constants:

```
>78
>F
>37AC
```

2.3.3 CHARACTER CONSTANTS

A character constant is written as a string of one or two characters enclosed in single quotes. For each single quote required within a character constant, two consecutive single quotes are required to represent the quote. The characters are represented internally as eight-bit ASCII characters, with leading bit equal to zero. A character constant consisting only of two single quotes (no character) is valid, and is assigned the value 0000_{16}.

The following are valid character constants:

Constant	Value
'AB'	4142_{16}
'C'	0043_{16}
'N'	$004E_{16}$
'''D'	2744_{16}

2.3.4 ASSEMBLY-TIME CONSTANTS

An assembly-time constant is written as an expression in the operand field of an EQU directive (paragraph 5.3.4). Any symbol in the expression must have been previously defined. The value of the label is determined at assembly time, and is absolute or relocatable as defined in paragraph 4.2.

2.4 SYMBOLS

Symbols are used in the label field, the operator field, and the operand field. A symbol is a string of alphanumeric characters, the first of which must be an alphabetic character, and none of which may be a blank. When more than six characters are used in a symbol, the assembler prints all the characters, but accepts only the first six characters for processing. User-defined symbols are valid only during the assembly in which they are defined.

When a symbol is used in the label field, it is associated with a location in the program, and must not be used as a label in any other statement. The mnemonic operation codes and the assembler directive names are valid user-defined symbols when placed in the label field.

The DXOP directive (paragraph 5.5.1) defines a symbol to be used in the operator field. No other user-defined symbol may be used in the operator field. Any symbol that is used in the operand field must be placed in the label field of a statement, or in the operand field of a REF directive (paragraph 5.4.2) with two exceptions. One exception is the operand field of the DXOP directive (paragraph 5.5.1). The other exception is the dollar sign character ($) used in expressions to represent the current location within the program (HERE).

The following are examples of valid symbols:

```
START

A1

OPERATION

$
```

2.5 TERMS

Terms are used in the operand fields of machine instructions and an assembler directive. A term is a decimal or hexadecimal constant, an absolute assembly-time constant, or an absolute label.

The following are examples of valid terms:

```
12

>C

WR2
```

Note that WR2 is valid as a term only if it has an absolute value. If START were a relocatable symbol and WR2 were defined as follows, WR2 would be relocatable, and not a valid term:

```
WR2  EQU  START+4
```

2.6 CHARACTER STRINGS

Several assembler directives require character strings in the operand field. A character string is written as a string of characters enclosed in single quotes. For each single quote in a character string, two consecutive single quotes are required to represent the single quote within the character string. The maximum length of the string is defined for each directive that requires a character string. The characters are represented internally as eight-bit ASCII characters.

The following are valid character strings:

```
'SAMPLE PROGRAM'

'PLAN ''C'''

'OPERATOR MESSAGE * PRESS START SWITCH'
```

SECTION III

ADDRESSING MODES

3.1 GENERAL

Five addressing modes are available for either operand of Format I instructions, and for the source operand of Format III, Format IV, Format VI, and Format IX instructions. The addressing modes are summarized in table 3-1 and are described in the following paragraphs.

Table 3-1. Addressing Modes

Addressing Mode	T field value (Note 1)	Example	Note
Workspace Register	0	5	
Workspace Register Indirect	1	*7	
Symbolic Memory	2	@LABEL	2, 3
Indexed Memory	2	@LABEL(5)	2, 4
Workspace Register Indirect Autoincrement	3	*7+	

Notes:

1. The T field is described in table 9-1.
2. The instruction requires an additional word for each T field value of 2. This word contains a memory address.
3. The S or D field is set to zero by the assembler.
4. Workspace register 0 cannot be used for indexing.

3.1.1 WORKSPACE REGISTER ADDRESSING

Workspace register addressing specifies a workspace register that contains the operand. A workspace register address is written as a term having a value in the range of 0 to 15.

The following example shows a MOV instruction and a COC instruction having two workspace register addresses each. A workspace register may be assigned a symbolic address as in the second example. Use an EQU directive having an appropriate absolute value.

```
MOV   4,>8
COC   15,R10
```

3.1.2 WORKSPACE REGISTER INDIRECT ADDRESSING

Workspace register indirect addressing specifies a workspace register that contains the address of the operand. An indirect workspace register address is written as a term preceded by an asterisk (*).

The following example shows two MOV instructions. The first instruction moves a word at the address in workspace register 7 to the address in workspace register 2. The second instruction moves a word at the address in workspace register 7 to workspace register 0.

```
MOV   *7,*2
MOV   *7,0
```

3.1.3 SYMBOLIC MEMORY ADDRESSING

Symbolic memory addressing specifies a memory address that contains the operand. A symbolic memory address is written as an expression preceded by an at sign (@).

The following example shows three MOV instructions. The first instruction moves a word from the address assigned to TABLE1 to the address assigned to LIST4. The second instruction moves the contents of workspace register 0 to the address assigned to STORE. The third instruction moves the contents of address $000C_{16}$ to address $007C_{16}$.

```
MOV   @TABLE1,@LIST4
MOV   0,@STORE
MOV   @12,@>7C
```

3.1.4 INDEXED MEMORY ADDRESSING

Indexed memory addressing specifies a memory address that contains the operand. The address is the sum of the contents of a workspace register and a symbolic address. An indexed memory address is written as an expression preceded by an at sign (@) and followed by a term enclosed in parentheses. The workspace register specified by the term within parentheses is the index register. Workspace register 0 may not be specified as an index register.

The following example shows two MOV instructions. The first instruction moves a word at a memory address to workspace register 6. The memory address is the sum of 2 and the contents of workspace register 7. The second instruction moves the contents of workspace register 7 to a memory address.

The memory address is the result of subtracting 6 from the location assigned to LIST4 and adding the contents of workspace register 5 to the difference.

```
MOV   @2(7), 6
MOV   7,@LIST4-6(5)
```

3.1.5 WORKSPACE REGISTER INDIRECT AUTOINCREMENT ADDRESSING

Workspace register indirect autoincrement addressing specifies a workspace register that contains the address of the operand. After the address is obtained from the workspace register, the workspace register is incremented. The workspace register increment is one for byte operations and two for word operations. A workspace register autoincrement address is written as a term preceded by an asterisk (*) and followed by a plus sign (+).

The following example shows a MOV instruction that moves a word at the address in workspace register 3 to workspace register 2. Then the contents of workspace register 3 is incremented by 2.

```
MOV   *3+, 2
```

3.2 PROGRAM COUNTER RELATIVE ADDRESSING

Program counter relative addressing is used by the Jump Instructions of Format II. A program counter relative address is an expression that corresponds to a byte address. The assembler evaluates the expression and subtracts the sum of location counter value plus two. One-half of the difference is the value that is placed in the object code. This value must be in the range of -128 to +127. The following example shows a program counter relative address:

```
JMP THERE
```

When the instruction is in relocatable code, the expression must be relocatable. When the instruction is in absolute code, the expression must be absolute.

3.3 CRU BIT ADDRESSING

The CRU Bit Instructions use an expression that represents a displacement from the CRU address contained in bits 3 through 14 of workspace register 12. The displacement, in the range of -128 to +127, is added algebraically to the contents of workspace register 12. The following examples show CRU bit addresses:

```
SBO   8
SBO   DTR
```

When DTR has been assigned the value of 8 by an EQU directive, paragraph 5.3.4, the two instructions would be equivalent, and would cause bit 8

relative to the CRU base address in workspace register 12 to be set to a logic one.

3.4 IMMEDIATE ADDRESSING

Immediate instructions use the contents of the word following the instruction word as an operand of the instruction. The immediate value is an expression, and the value of the expression is placed in the word following the instruction by the assembler. Those immediate instructions that require two operands have a workspace register address preceding the immediate value. The following examples show an immediate address in an instruction that requires an immediate operand only, and in an instruction that requires two operands:

```
LIMI   5
LI     5,>1000
```

SECTION IV

RELOCATABILITY

4.1 RELOCATION OF CODE

The Model 990 Assembler assembles both absolute and relocatable object code. Absolute object code is code that must be placed in specified memory locations and is appropriate for programs that occupy dedicated areas of memory. Relocatable object code is code that may be placed in any available locations. All relocatable address information must be modified for the actual memory locations in which the program is placed. Relocatability allows programs to share memory in many possible combinations.

4.2 RELOCATABILITY OF SOURCE STATEMENT ELEMENTS

Elements of source statements are expressions, constants, symbols, and terms. Terms are absolute in all cases; the other elements may be either absolute or relocatable.

The relocatability of an expression is a function of the relocatability of the symbols and constants that make up the expression. An expression is relocatable when it contains one or more relocatable constants or symbols, and the number of relocatable symbols or constants added to the expression is one greater than the number of relocatable symbols or constants subtracted from the expression. (All other valid expressions are absolute). When the first symbol or constant is unsigned, it is considered to be added to the expression. When a unary minus follows an addition operator in an expression, the effective operation is subtraction. When a unary minus follows a subtraction operator, the effective operation is addition. For example, when all symbols in the following expressions are relocatable, the expressions are relocatable:

LABEL--1

LABEL+TABLE+-INC

-LABEL+TABLE+INC

Decimal, hexadecimal, and character constants are absolute. Assembly-time constants defined by absolute expressions are absolute, and assembly-time constants defined by relocatable expressions are relocatable.

Any symbol that appears in the label field of a source statement other than an EQU directive (paragraph 5.3.4) is absolute when the statement is in an absolute block of the program. Any symbol that appears in the label field of a source statement other than an EQU directive is relocatable when the statement is in a relocatable block of the program.

A location may be defined as absolute (paragraph 5.1.1) or as relocatable (paragraph 5.1.2). The location may contain either an absolute or relocatable values. The sample program in appendix B includes absolute locations with relocatable contents and relocatable locations with absolute contents.

SECTION V

ASSEMBLER DIRECTIVES

5.1 DIRECTIVES AFFECTING THE LOCATION COUNTER

Five assembler directives affect only the location counter of the assembler. Two of these also define the succeeding block of the program as absolute or relocatable. The location counter is a component of the assembler that contains the present location.

Until an Absolute Origin directive is processed by the assembler, the location counter contents are relocatable. Subsequent Relocatable Origin directives cause the location counter to be set to the specified relocatable value, and to continue assembling relocatable object code. This concatenates all relocatable blocks within an assembly into a single relocatable segment. The total length of this segment is the length of the relocatable code assembled.

The Block Starting with Symbol and Block Ending with Symbol directives advance the location counter, forming an area for storage of data. The Word Boundary directive aligns the location counter to a word boundary (even address).

5.1.1 ABSOLUTE ORIGIN (AORG)

AORG places a value in the location counter and defines the succeeding locations as absolute. Use of the label field is optional. When a label is used, it is assigned the value that the directive places in the location counter. The operator field contains AORG. The operand field contains a well-defined expression. The assembler places the value of the well-defined expression in the location counter. Use of the comment field is optional.

The following example shows an AORG directive:

```
AORG   >1000+X
```

Symbol X must be absolute and must have been previously defined. If X has a value of 6, the location counter is set to 1006_{16} by this directive. Had a label been included, the label would have been assigned the value 1006_{16}.

5.1.2 RELOCATABLE ORIGIN (RORG)

RORG places a value in the location counter and defines the succeeding locations as relocatable. Use of the label field is optional. When a label is used, it is assigned the value that the directive places in the location counter. The operator field contains RORG. The operand field is optional , and when the operand field is not used, zero or the value that was in the location counter following assembly of the preceding relocatable location is placed in the location counter. When the operand field is used, a relocatable expression that

contains no symbols not previously defined is placed in the operand field. The comment field may be used only when the operand field is used.

The following example shows an RORG directive:

```
RORG   $-20   OVERLAY TEN WORDS
```

The $ symbol refers to the location following the preceding relocatable location of the program. This has the effect of backing up the location counter ten words. The instructions and directives following the RORG directive replace the ten previously assembled words of relocatable code, permitting correction of the program without removing source records. Had a label been included, the label would have been assigned the value placed in the location counter. An example of a RORG directive with no operand field is as follows:

```
SEG2   RORG
```

Assume that after defining data for a program, which occupied 44_{16} bytes, an AORG directive initiated an absolute block of code. The absolute block is followed by the RORG directive in the above example, which places 0044_{16} in the location counter and defines the location counter as relocatable. Symbol SEG2 is a relocatable value, 0044_{16}. The RORG directive in the above example would have no effect except at the end of an absolute block.

5.1.3 BLOCK STARTING WITH SYMBOL (BSS)

BSS assigns the value in the location counter to the symbol in the label field and advances the location counter according to the value in the operand field. The label field contains the label of the first byte in the block. The operator field contains BSS. The operand field contains a well-defined expression that represents the number of bytes to be added to the location counter. The comment field is optional.

The following example shows a BSS directive:

```
BUFF1   BSS 80   CARD INPUT BUFFER
```

This directive reserves an 80-byte buffer at location BUFF1.

5.1.4 BLOCK ENDING WITH SYMBOL (BES)

BES advances the location counter according to the value in the operand field and assigns the new location counter value to the symbol in the label field. The label field contains the label of the location following the block. The operator field contains BES. The operand field contains a well-defined expression that represents the number of bytes to be added to the location counter. The comment field is optional.

The following example shows a BES directive:

```
BUFF2   BES   >10
```

The directive reserves a 16-byte buffer. Had the location counter contained 100_{16} when the assembler processed this directive, BUFF2 would have been assigned the value 110_{16}.

5.1.5 WORD BOUNDARY (EVEN)

EVEN places the location counter on the next word boundary (even) byte address. When the location counter is already on a word boundary, the location counter is not altered. Use of the label field is optional. When a label is used, the value in the location counter after processing the directive is assigned to the label. The operator field contains EVEN. The operand field is not used, and the comment field is optional.

The following example shows an EVEN directive:

```
WRF1    EVEN    WORKSPACE REGISTER FILE ONE
```

The directive assures that the location counter contains a word boundary address, and assigns that address to label WRF1. Use of an EVEN directive preceding or following a machine instruction or a DATA directive (paragraph 5.3.2) is redundant. The assembler advances the location counter to an even address when it processes a machine instruction or a DATA directive.

5.2 DIRECTIVES AFFECTING ASSEMBLER OUTPUT

Five assembler directives affect assembler output. One affects the object code output of the assembler and the remaining four affect the source listing output of the assembler.

The Program Identifier directive supplies a program name, which is placed in the object code for use by the linking loader.

The Page Title directive supplies a title to be printed at the top of each page of the source listing. The List Source directive restores printing of the source listing when printing has been inhibited by a No Source List directive. The Page Eject directive causes the assembler to print a heading and continue the source listing on a new page.

5.2.1 PROGRAM IDENTIFIER (IDT)

IDT assigns a name to the program. An IDT directive must precede any machine instruction or assembler directive that results in object code. Use of the label field is optional. When a label is used, the current value of the location counter is assigned to the label. The operator field contains IDT. The operand field contains the program name, a character string of up to eight characters. When a character string of more than eight characters is entered, the assembler prints a truncation error message, and retains the first eight characters as the program name. The comment field is optional.

The following example shows an IDT directive:

```
IDT   'CONVERT'
```

The directive assigns the name CONVERT to the program to be assembled. The program name is printed in the source listing as the operand of the IDT directive, but does not appear in the page heading of the source listing. The program name is placed in the object code, but serves no purpose during the assembly.

5.2.2 PAGE TITLE (TITL)

TITL supplies a title to be printed in the heading of each page of the source listing. When a title is desired in the heading of the first page of the source listing, a TITL directive must be the first source statement submitted to the assembler. This directive is not printed in the source listing. Use of the label field is optional. When a label is used, the current value of the location counter is assigned to the label. The operator field contains TITL. The operand field contains the title, a character string of up to 50 characters. When more than 50 characters are entered, the assembler retains the first 50 characters as the title, and prints a truncation error message. The comment field is optional, but the assembler does not print the comment.

The following example shows a TITL directive:

```
TITL   '** REPORT GENERATOR **'
```

The directive causes the title ** REPORT GENERATOR ** to be printed in the page headings of the source listing. When a TITL directive is the first source statement in a program, the title is printed on all pages until another TITL directive is processed. Otherwise, the title is printed on the next page after the directive is processed, and on subsequent pages until another TITL directive is processed.

5.2.3 LIST SOURCE (LIST)

LIST restores printing of the source listing. This directive is required only when a No Source List directive is in effect, to cause the assembler to resume listing. This directive is not printed in the source listing. Use of the label field is optional. When a label is used, the current value of the location counter is assigned to the label. The operator field contains LIST. The operand field is not used. Use of the comment field is optional, but the assembler does not print the comment.

The following example shows a LIST directive:

```
LIST
```

The directive causes the source listing to be resumed with the next source statement.

5.2.4 NO SOURCE LIST (UNL)

UNL inhibits printing of the source listing. The UNL directive is not printed in the source listing. Use of the label field is optional. When a label is used, the current value of the location counter is assigned to the label. The operator field contains UNL. The operand field is not used. Use of the comment field is optional, but the assembler does not print the comment.

The following example shows UNL directive:

```
UNL
```

The directive inhibits printing of the source listing. Use of the UNL directive to inhibit printing reduces assembly time and the size of the source listing.

5.2.5 PAGE EJECT (PAGE)

PAGE causes the assembler to continue the source program listing on a new page. The PAGE directive is not printed in the source listing. Use of the label field is optional. When a label is used, the current value of the location counter is assigned to the label. The operator field contains PAGE. The operand field is not used. Use of the comment field is optional, but the assembler does not print the comment.

The following example shows a PAGE directive:

```
PAGE
```

The directive causes the assembler to begin a new page of the source listing. The next source statement is the first statement listed on the new page. Use of the Page directive to begin new pages of the source listing at the logical divisions of the program improves documentation of the program.

5.3 DIRECTIVES THAT INITIALIZE CONSTANTS

Four assembler directives assign initial values to constants. The Initialize Byte directive initializes one or more bytes of memory with eight-bit two's complement numbers. The Initialize Word directive initializes one or more words of memory with 16-bit two's complement numbers. The Initialize Text directive places ASCII characters in successive bytes of memory. The Define Assembly-Time Constant directive assigns a value to a symbol.

5.3.1 INITIALIZE BYTE (BYTE)

BYTE places one or more values in one or more successive bytes of memory. Use of the label field is optional. When a label is used, the location at which the assembler places the first byte is assigned to the label. The operator field contains BYTE. The operand field contains one or more expressions separated by commas. The expressions must contain no symbols that are

not previously defined and no external references. The assembler evaluates each expression and places the value in a byte as an eight-bit two's complement number. When truncation is required, the assembler prints a truncation error message and places the rightmost portion of the value in the byte. The comment field is optional.

The following example shows a BYTE directive:

```
KONS    BYTE    >F+1,-1,'D'-'=',0,'AB'-'AA'
```

The directive initializes five bytes, starting with a byte at location KONS. The contents of the resulting bytes is 00010000, 11111111, 00000111, 00000000, and 00000001.

5.3.2 INITIALIZE WORD (DATA)

DATA places one or more values in one or more successive words of memory. The assembler advances the location counter to a word boundary (even) address. Use of the label field is optional. When a label is used, the location at which the assembler places the first word is assigned to the label. The operator field contains DATA. The operand field contains one or more expressions separated by commas. The assembler evaluates each expression and places the value in a word as a sixteen-bit two's complement number. The comment field is optional.

The following example shows a DATA directive:

```
KONS1   DATA    3200,1+'AB',-'AF',>F4A0,'A'
```

The directive initializes five words, starting with a word at location KONS1. The contents of the resulting words are $0C80_{16}$, 4143_{16}, $BEBA_{16}$, $F4A0_{16}$, and 0041_{16}. Had the location counter contents been $010F_{16}$ prior to processing this directive, the value assigned to KONS1 would be 0110_{16}.

5.3.3 INITIALIZE TEXT (TEXT)

TEXT places one or more characters in successive bytes of memory. The assembler negates the last character of the string when the string is preceded by a minus (-) sign (unary minus). Use of the label field is optional. When a label is used, the location at which the assembler places the first character is assigned to the label. The operator field contains TEXT. The operand field contains a character string of up to 52 characters, which may be preceded by a unary minus sign. The comment field is optional.

The following example shows a TEXT directive:

```
MSG1    TEXT 'EXAMPLE'    MESSAGE HEADING
```

The directive places the eight-bit ASCII representations of the characters in successive bytes. When the location counter is on an even address, the result, in hexadecimal representation, is 4558, 414D, 504C, and 45XX. XX represents the contents of the rightmost byte of the fourth word, which are

determined by the next source statement. The label MSG1 is assigned the value of the first byte address in which 45 is placed. Another example, showing the use of a unary minus, is as follows:

```
MSG2  TEXT  -"NUMBER"
```

When the location counter is on an even address, the result, in hexadecimal representation, is 4E55, 4D42, and 45AE. The label MSG2 is assigned the value of the byte address in which 4E is placed.

5.3.4 DEFINE ASSEMBLY-TIME CONSTANT (EQU)

EQU assigns a value to a symbol. The label field contains the symbol. The operator field contains EQU. The operand field contains an expression in which all symbols have been previously defined. Use of the comment field is optional.

The following example shows an EQU directive:

```
R0   EQU  0   WORKSPACE REGISTER 0
```

The directive assigns an absolute value to the symbol R0, making R0 available to use as a workspace register address. Another example of an EQU directive is:

```
TIME   EQU   HOURS
```

The directive assigns the value of previously defined symbol HOURS to symbol TIME. When HOURS appears in the label field of a machine instruction in a relocatable block of the program, the value is a relocatable value. The two symbols may be used interchangeably.

5.4 DIRECTIVES THAT LINK PROGRAMS

Two assembler directives provide links between programs that are assembled separately. The External Definition directive makes one or more symbols in a program available to other programs. The External Reference directive provides access to one or more symbols from other programs for use in a program. The programs may be linked and executed as one program.

5.4.1 EXTERNAL DEFINITION (DEF)

DEF makes one or more symbols available to other programs for reference. The use of the label field is optional. When a label is used, the current value of the location counter is assigned to the label. The operator field contains DEF. The operand field contains one or more symbols, separated by commas, to be defined in the program being assembled. The comment field is optional.

The following example shows a DEF directive:

```
DEF   ENTER,ANS
```

The directive causes the assembler to include symbols ENTER and ANS in the object code so that these symbols are available to other programs. When the DEF directive does not precede the source statements that contain the symbols, the assembler identifies the symbols as multiply defined symbols.

5.4.2 EXTERNAL REFERENCE (REF)

REF provides access to one or more symbols defined in other programs. The use of the label field is optional. When a label is used, the current value of the location counter is assigned to the label. The operator field contains REF. The operand field contains one or more symbols, separated by commas, to be used in the operand field of a subsequent source statement. The comment field is optional.

The following example shows a REF directive:

```
REF   ARG1,ARG2
```

The directive causes the assembler to include symbols ARG1 and ARG2 in the object code so that the corresponding addresses may be obtained from other programs.

NOTE

An external reference will not be inserted by the loader at absolute location 0.

5.5 MISCELLANEOUS DIRECTIVES

Two miscellaneous directives are available. The Define Extended Operation directive assigns a symbol for an extended operation. The Program End directive terminates the source program.

5.5.1 DEFINE EXTENDED OPERATION (DXOP)

DXOP assigns a symbol to be used in the operator field to specify an extended operation. The use of the label field is optional. When a label is used, the current value in the location counter is assigned to the label. The operator field contains DXOP. The operand field contains a symbol followed by a comma and a term. The symbol assigned to an extended operation must not be used in the label or operand field of any other statement. The assembler assigns the symbol to an extended operation specified by the term, which must have a value in the range of 0 to 15. The comment field is optional.

The following example shows a DXOP directive:

```
DXOP   DADD,13
```

The directive defines DADD as extended operation 13. When the assembler recognizes the symbol DADD in the operator field, it assembles an XOP instruction (paragraph 6.9.1) that specifies extended operation 13. The XOP

instruction is described in the Model 990 Reference Manual. The following example shows the use of the symbol DADD in a source statement:

```
DADD  @LABEL1(4)
```

The assembler places the operand field contents in the T_S and S fields of an XOP instruction, and places 13 in the D field.

5.5.2 PROGRAM END (END)

END terminates the assembly. The last source statement of a program is the END directive. When any source statements follow the END directive, they are ignored. Use of the label field is optional. When a label is used, the current value in the location counter is assigned to the symbol. The operator field contains END. Use of the operand field is optional. When the operand field is used, it contains a symbol that specifies the entry point of the program. When the operand field is not used, no entry point is placed in the object code. The comment field may be used only when the operand field is used.

The following example shows an END directive:

```
END   START
```

The directive causes the assembler to terminate the assembly of this program. The assembler also places the value of START in the object code as an entry point.

When a program executes in a stand-alone mode, and is loaded by the ROM loader, it must supply an entry point to the loader. When no operand is included in the END directive, and that program is loaded by the ROM loader, the loader transfers control to the entry point of the loader, and attempts to load another object program.

When a program is to be loaded by the Linking Loader (LAL990) the END directive does not require an operand unless the program is to be loaded and linked to other programs and contains the entry point for the resulting linked program. LAL990 returns control to the first relocatable location when the program or programs loaded do not specify entry points. When LAL990 loads a set of programs, and more than one of these programs specifies an entry point, LAL990 transfers control to the last entry point it receives.

SECTION VI

MACHINE INSTRUCTIONS

6.1 GENERAL

This section describes the source statement formats for machine instructions. The operation of each machine instruction is described in the Model 990 Reference Manual. There are nine formats of machine code, shown in figure 10-3. The source statement formats correspond to the machine code formats, except that two of the machine code formats each require more than one source statement format.

Source statements that contain machine instructions use the label field, the operator field, the operand field, and the comment field defined in paragraph 2.1. Use of the label field is optional for machine instructions. When the label field is used, the label is assigned the address of the machine instruction. The assembler advances the location counter to a word boundary (even address) before assembling a machine instruction. The operator field contains the mnemonic operation code of the instruction. The contents of the operand field is defined for each format in the following paragraphs. The use of the comment field is optional.

In the descriptions of source statement formats in the following paragraphs, a general address is in one of the five addressing modes described in Section III. A workspace register address is the workspace register address described in paragraph 3.2.

6.2 FORMAT I - TWO ADDRESS INSTRUCTIONS

The operand field of Format I instructions contains two general addresses separated by a comma. The first address is the source address; the second is the destination address. The following mnemonic operation codes use Format I:

A	MOV	SOC
AB	MOVB	SOCB
C	S	SZC
CB	SB	SZCB

The following example shows a source statement for a Format I instruction:

```
SUM   A   @LABEL1,*7
```

The label SUM refers to the location at which the assembler places the instruction. The operator field specifies an add words instruction. The sum of the word at location LABEL1 and the word at the address contained in workspace register 7 is placed in the address contained in workspace register 7.

6.3 FORMAT II JUMP INSTRUCTIONS

The operand field of Format II Jump Instructions contains an expression that corresponds to a byte address. When the byte address is not on a word boundary (an even address), the assembler subtracts one to obtain a word boundary address. When the instruction is in an absolute block of a program, the expression in the operand field must be absolute. When the instruction is in a relocatable block, the expression must be relocatable.

The assembler adds two to the location counter contents and subtracts the sum from the address corresponding to the expression in the operand field. The assembler divides the difference by two to obtain a displacement in words. The displacement must be in the range of -128 to +127.

The following mnemonic operation codes are Format II Jump Instructions:

```
JEQ   JLE   JNE
JGT   JLT   JNO
JH    JMP   JOC
JHE   JNC   JOP
JL
```

The following example shows a source statement for a Format II Jump Instruction:

```
JMP   BEGIN
```

The label field is not used. The operator field specifies a jump unconditional instruction. Control transfers to the instruction at location BEGIN.

6.4 FORMAT II DIGITAL INPUT/OUTPUT BIT INSTRUCTIONS

The operand field of Format II Digital Input/Output Bit Instructions contains a well-defined expression. The value of the expression is a bit address relative to a base address in workspace register 12. The value of the expression must be in the range of -128 to +127. The following mnemonic operation codes are Format II Digital Input/Output Instructions:

```
SBO   SBZ   TB
```

The following example shows a source statement for a Format II Digital Input/Output Instruction:

```
SBO   5
```

The label field is not used. The operator field specifies a set bit to one instruction. The operand field specifies bit 5 relative to a base address in workspace register 12. Assuming that the base address has been set to the lowest address of a group of CRU bits connected to a digital input/output address, this instruction sets bit 5 of the group to one.

6.5 FORMAT III - LOGICAL INSTRUCTIONS

The operand field of Format III instructions contains a general address followed by a comma and a workspace register address. The general address is the source address. The workspace register address is the destination address. The following mnemonic operation codes use Format III:

COC CZC XOR

The following example shows a source statement for a Format III instruction:

```
COMP   XOR @LABEL8(3), 5
```

The label COMP refers to the location at which the assembler places the instruction. The operator field specifies an exclusive OR instruction. The result of an exclusive OR operation between the contents of a word at location LABEL8 indexed by workspace register 3 and the contents of workspace register 5 is placed in workspace register 5.

6.6 FORMAT IV - CRU INSTRUCTIONS

The operand field of Format IV instructions contains a general address followed by a comma and a term. The general address is the memory address from which or into which bits will be transferred. The CRU address for the transfer is the contents of workspace register 12. The term is the number of bits to be transferred, and must have a value in the range of 0 to 15 (a 0 value transfers 16 bits). The following mnemonic operation codes use Format IV:

LDCR STCR

The following example shows a source statement for a Format IV instruction:

```
LDCR   *6+, 8
```

The label field is not used. The operator field specifies a load communication register instruction. The instruction loads a byte from the byte address in workspace register 6 into the CRU at the location in workspace register 12, and increments the address in workspace register 6 by one.

6.7 FORMAT V - REGISTER SHIFT INSTRUCTIONS

The operand field of Format V instructions contains a workspace register address followed by a comma and a term. The contents of the workspace register are shifted a number of bit positions specified by the term. When the term equals zero, the shift count must be placed in bits 12-15 of workspace register 0. The value of the term must be in the range of 0 to 15. The following mnemonic operation codes use Format V:

SLA SRC SRL
SRA

The following example shows a source statement for a Format V instruction:

```
SLA    6, 4
```

The label field is not used. The operator field specifies a shift left arithmetic instruction. The instruction shifts the contents of workspace register 6 to the left four bit positions.

6.8 FORMAT VI - SINGLE ADDRESS INSTRUCTIONS

The operand field of Format VI instructions contains a general address. The following mnemonic operation codes use Format VI:

ABS	DEC	NEG
B	DECT	SETO
BL	INC	SWPB
BLWP	INCT	X
CLR	INV	

The following example shows a source statement for a Format VI instruction:

```
CNT   INC   7
```

The label CNT refers to the location at which the assembler places the instruction. The operator field specifies an increment instruction. The instruction adds one to the contents of workspace register 7 and places the sum in workspace register 7.

6.9 FORMAT VII - CONTROL INSTRUCTIONS

Format VII instructions require no operand field. The following mnemonic operation codes use Format VII:

CKOF	IDLE	RSET
CKON	LREX	RTWP

The following example shows a source statement for a Format VII instruction:

```
RTWP   RETURN TO MAIN PROGRAM
```

The label field is not used. The operator field specifies a return from interrupt subroutine instruction. The comment field follows the operator field because no operand field is required.

6.10 FORMAT VIII - IMMEDIATE INSTRUCTIONS

The operand field of Format VIII instructions contains a workspace register address followed by a comma and an expression. The workspace register address is the destination address, and the expression is the immediate operand. The following mnemonic operation codes use Format VIII:

AI	LI
ANDI	ORI
CI	

The following example shows a source statement for a Format VIII instruction:

```
ANDI   4,>000F
```

The label field is not used. The operator field specifies an AND immediate instruction. The instruction performs an AND operation with the contents of workspace register 4 and the number $000F_{16}$. The effect is to mask out the 12 leftmost bits of the workspace register contents.

Two Format VIII instructions require only an expression in the operand field. The expression is the immediate operand. The destination address is implied in the name of the instruction. The following mnemonic operation codes use this modified Format VIII:

```
LIMI   LWPI
```

Another example shows this modified Format VIII:

```
LWPI   WRK1
```

The label field is not used. The operator field specifies a load workspace pointer immediate instruction. The location that corresponds to label WRK1 is placed in the WP register.

Two other Format VIII instructions require only a workspace register address in the operand field. The workspace register address is the destination address. The source is implied in the name of the instruction. The following mnemonic operation codes use this modified Format VIII:

```
STST   STWP
```

The following example shows a source statement for a Store Workspace Pointer instruction:

```
STWP   4
```

The label field is not used. The operator field specifies a store workspace pointer instruction. The operand field specifies workspace register 4. The instruction transfers the contents of the workspace pointer into workspace register 4.

6.11 FORMAT IX - EXTENDED OPERATION INSTRUCTION

The operand field of a Format IX Extended Operation instruction contains a general address and a term. The general address is the address of the operand for the extended operation. The term specifies the extended operation to be performed and must be in the range of 0 to 15. The mnemonic operation code is XOP.

The following example shows a source statement for a Format IX Extended Operation instruction:

```
XOP   @LABEL(4), 12
```

The label field is not used. The operator field specifies an extended operation instruction. The operand field specifies that extended operation 12 is to be performed with the contents of a word at location LABEL indexed by workspace register 4. The DXOP directive (paragraph 5.5.1) can be used to define an extended operation.

6.12 FORMAT IX MULTIPLY AND DIVIDE INSTRUCTIONS

The operand field of Format IX Multiply and Divide instructions contains a general address followed by a comma and a workspace register address. The general address is the address of the multiplier or divisor, and the workspace register address is the address of the workspace register that contains the multiplicand or dividend. The workspace register address is also the address of the first of two workspace registers to contain the result. The mnemonic operation codes are MPY and DIV.

The following example shows a source statement for a Format IX Multiply instruction:

```
MPY   @ACC,9
```

The label field is not used. The operator field specifies a multiply instruction. The operand field instruction multiplies the contents of a word at location ACC by the contents of workspace register 9, and places the product in workspace registers 9 and 10.

SECTION VII

PSEUDO-INSTRUCTIONS

7.1 GENERAL

The Model 990 Assembly Language includes two pseudo-instructions, which are predefined symbols that cause the assembler to assemble certain machine instructions with specific operands. A pseudo-instruction is a convenient way to code an operation that is actually performed by a machine instruction. The pseudo-instructions are the No Operation and the Return instructions.

7.2 NO OPERATION (NOP)

NOP places a machine instruction in the object code which has no effect on execution of the program. Use of the label field is optional. When the label field is used, the label is assigned the location of the instruction. The operator field contains NOP. The operand field is not used. Use of the comment field is optional.

Enter the NOP pseudo-instruction as shown in the following example:

```
MOD    NOP
```

Location MOD contains a NOP pseudo-instruction when the program is loaded. Another instruction may be placed in location MOD during execution to implement a program option. The assembler supplies the same object code as if the source statement had contained the following:

```
MOD    JMP    $+2
```

7.3 RETURN (RT)

RT places a machine instruction in the object code to return control to a calling routine from a subroutine. Use of the label field is optional. When the label field is used, the label is assigned the location of the instruction. The operator field contains RT. The operand field is not used.

Use of the comment field is optional. Enter the RT pseudo-instruction as shown in the following example:

```
RT
```

The assembler supplies the same object code as if the source statement had contained the following:

```
B    *11
```

When control is transferred to a subroutine by execution of a BL instruction, the link to the calling routine is stored in workspace register 11. An RT

pseudo-instruction returns control to the instruction following the BL instruction in the calling routine.

SECTION VIII

SUBROUTINE CALLING AND RETURN

8.1 COMMON WORKSPACE SUBROUTINE

One type of subroutine supported by the Model 990 Assembly Language uses the same set of workspace registers that the calling routine uses. The BL instruction branches to a common workspace subroutine, and stores the return address in workspace register 11. The subroutine uses an RT pseudo-instruction to return control to the calling routine at the instruction following the BL instruction. A common workspace subroutine may use other branch instructions as appropriate to transfer control to other points in the calling routine or in other subroutines.

8.2 CONTEXT SWITCH SUBROUTINES

Another type of subroutine supported by the Model 990 Assembly Language consists of hardware interrupt subroutines, extended operation subroutines, and user subroutines. The method of branching to and returning from these subroutines is similar, and is called a context switch. The subroutine has a workspace which becomes the active workspace when the subroutine receives control. The environment of the calling or interrupted routine is stored. When the subroutine returns control to the calling or interrupted routine, the calling environment is restored.

When the user writes a hardware interrupt subroutine, he must place the workspace pointer and the entry point in the pair of memory words assigned to the level of the interrupt. Multiply the interrupt level number by four to obtain the address of this pair of memory words. The subroutine workspace pointer must be placed in the first word, and the subroutine entry point must be placed in the second word. The workspace pointer of the interrupted program is stored in workspace register 13 of the subroutine workspace. The return address is stored in workspace register 14. The Status Register contents at interrupt time is stored in workspace register 15. The subroutine returns control to the interrupted program at the interrupt point with an RTWP instruction. The instruction restores the interrupted environment as it returns control to the instruction following the interrupt point.

When the user writes an extended operation subroutine, he must place the workspace pointer and the entry point in the pair of memory words assigned to the extended operation. An extended operation is specified by a number, 0 to 15, in the XOP instruction. Multiply the number by four and add the product to 40_{16}. This is the address of the pair of words assigned to the extended operation. The subroutine workspace pointer must be placed in the first word, and the subroutine entry point must be placed in the second word. The use of workspace registers described in the preceding paragraph applies. An extended operation subroutine is entered by executing an XOP instruction

that specifies the operation. The subroutine returns control to the calling routine with an RTWP instruction. The instruction restores the calling routine environment and returns control at the instruction following the XOP instruction.

A user context switch subroutine is entered by a BLWP instruction. The operand of the instruction is the address of a pair of words that contain the subroutine workspace pointer and the subroutine entry point. An RTWP instruction restores the environment of the calling program and returns control at the instruction following the BLWP instruction. The use of workspace registers is the same as previously described for a hardware interrupt subroutine.

The details of the context switch resulting from execution of a BLWP instruction are shown in figure 8-1. Context switching because of an interrupt or an XOP instruction is similar to this example.

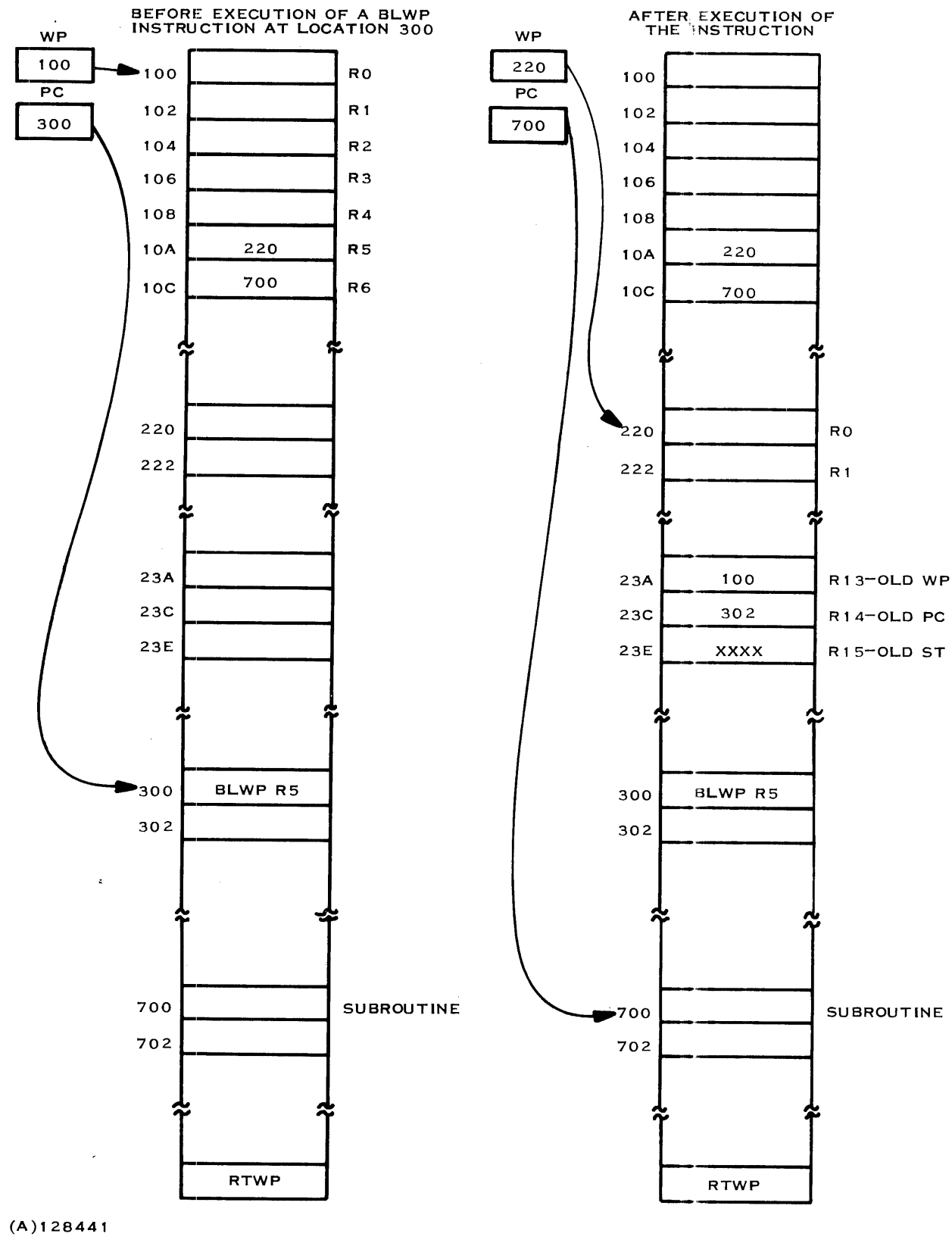


Figure 8-1. Context Switching

SECTION IX

PROGRAM MODULES

9.1 GENERAL

Since the assembler includes directives that generate the information required to link program modules, it is not necessary to assemble an entire program in the same assembly. A long program may be divided into separately assembled modules to avoid a long assembly or to reduce the symbol table size. Also, modules common to several programs may be combined as required. The linking loader links the programs as it loads them, so that the loaded program functions as if it had been assembled in a single assembly. The following paragraphs define the linking information that must be included in a program module.

9.2 EXTERNAL REFERENCE DIRECTIVE

Each symbol from another program module must be placed in the operand field of an REF directive in the program module that requires the symbol. The IDT character string of each program module that defines one or more of these symbols must also be placed in the operand field of an REF directive within one of the program modules being linked. The first module may contain an REF directive that contains the IDT character strings of all modules to be linked.

9.3 EXTERNAL DEFINITION DIRECTIVE

Each symbol defined in a program module and required by one or more other program modules must be placed in the operand field of a DEF directive.

9.4 PROGRAM IDENTIFIER DIRECTIVE

Subsequent program modules after the first module loaded by the linking loader must include an IDT directive. The first six characters of the IDT character string must be unique with respect to other IDT character strings submitted to the loader during the loading of the program.

9.5 LINKING PROGRAM MODULES

The linking loader builds a list of symbols from REF directives as it loads the program modules. The loader matches symbols from DEF directives to the symbols in the reference list. The loader also matches the first six characters of IDT character strings with symbols in the reference list.

When object code for several program modules is on the same cassette or paper tape, and a program that requires only some of these modules is being

loaded, the loader ignores those program modules whose IDT character strings do not appear in the reference list of the loader. This allows program modules from several cassettes or paper tapes to be loaded without requiring the user to locate the required modules on the cassettes or paper tapes. However, it requires that all referencing modules precede the modules they reference in the sequence in which the loader loads the modules.

SECTION X

ASSEMBLER OUTPUT

10.1 SOURCE LISTING

The Model 990 Computer Assembler prints a source listing that shows the source statements and the resulting object code. Appendix B includes a listing example.

10.1.1 LISTING FORMAT

Each page of the source listing has a title line at the top of the page. Any title supplied by a TITL directive is printed on this line, and a page number is printed to the right of the title area. The printer skips a line below the title line, and prints a line for each source statement listed. The line for each source statement contains a source record number, a location counter value, object code assembled, and the source statement as entered. When a source statement results in more than one word of object code, the assembler prints the location counter value and object code on a separate line following the source statement for each additional word of object code. The source listing lines for a machine instruction source statement are shown in the following example:

```
0018  0156  C820     MOV   @INIT+3, @3
      0158  012B'
      015A  0003
```

The source record number, 0018 in the example, is a four-digit decimal number. Source records are numbered in the order in which they are entered, whether they are listed or not. The TITL, LIST, UNL, and PAGE directives are not listed, and source records between a UNL directive and a LIST directive are not listed. The difference between source record numbers printed indicates how many source records are not listed.

The next field on a line of the listing contains the location counter value, a hexadecimal value. In the example, 0156 is the location counter value. Not all directives affect the location counter, and those that do not affect the location counter leave this field blank. Specifically, of the directives that the assembler lists, the IDT, REF, DEF, DXOP, EQU, and END directives leave the location counter field blank.

The third field contains the hexadecimal representation of the object code placed in the location by the Assembler, C820 in the example. The apostrophe following the third field of the second line in the example indicates that the contents, 012B, is relocatable. All machine instructions and the BYTE, DATA, and TEXT directives use this field for object code. The EQU directive places the value corresponding to the label in the object code field.

The third field may contain two or four hyphens (-) instead of hexadecimal digits. This occurs when a forward reference determines the values of these digits. Later, when the forward reference is defined, the assembler prints an additional line in the listing following the statement that defines the forward reference. This line contains the location being resolved, two asterisks (**), and the contents. An error-free listing will include such a line for each location previously printed with hyphens in the contents.

The fourth field contains the first 52 characters of source statement as supplied to the assembler. Spacing in this field is determined by the spacing in the source statement. The four fields of source statements will be aligned in the listing only when they are aligned in the same character positions in the source statements or when tag characters are used.

The machine instruction used in the example specifies the symbolic memory addressing mode for both operands. This causes the instruction to occupy three words of memory, and three lines of the listing. The object code corresponds to the operands in the order in which they appear in the source statement.

When object code is punched on the ASR33, the object code is printed as it is punched. Since the listing is being printed on the same device, lines of object code are printed between the lines of the source listing.

10.1.2 ERROR CODES

The assembler prints an error code on a separate line of the listing when it detects an error. The error code is printed in the following format:

```
** ERR 1 - LOC 012E **
```

The error code is 1 and the error is at location $012E_{16}$. Error codes are listed in table 10-1. This particular message was printed at the end of the assembly and identified an undefined symbol at the specified location. The statement that contained the undefined symbol was a statement that allows forward references. The symbol was therefore not undefined until the assembler recognized an END statement without having recognized a statement defining the symbol. The error code line may be printed at any point, from the line immediately following the statement in error to lines following the END statement.

The assembler can accommodate a minimum of 150 symbols in a 4K memory configuration. When the assembler is unable to continue because the area of memory available for symbols and forward references has been filled, the assembler prints the following message:

```
** ABORT **
```

The user may divide the program into two or more modules and assemble them separately. Considerations for properly linking these modules are

described in Section IX. Alternatively, the user may shorten the symbols in the program and reassemble. Since shorter symbols use less space in the symbol table, the capacity of the symbol table is increased by using short symbols.

Following the last statement or error message, the assembler prints undefined symbols, if there are any, one symbol per line. The undefined symbol may correspond to one of several error codes, or may be a symbol in a DEF directive that does not also appear in the label field of a statement.

Table 10-1. Error Codes

Code	Description
1	Undefined symbol. A symbol in the operand field of the statement corresponding to the error location does not appear in the label field of a source statement, or in the operand field of a REF directive.
2	Syntax error. The statement corresponding to the error location contains a syntax error.
3	Illegal external reference. The statement corresponding to the error location contains an external reference (and an arithmetic operator) in an expression or an external reference to be placed in a field smaller than 16 bits.
4	Truncation error. The statement corresponding to the error location contains a number that is too large or a character string that is too long. The number may be the result of evaluating an expression. Relocatability of a term or expression may be in error.
5	Multiply defined symbol. A symbol in the statement corresponding to the error location has been previously referenced or defined.
6	Unrecognizable operator. Contents of the operator field of the statement corresponding to the error location is not a mnemonic operation code, a directive, or a name defined as an extended operation.
7	Illegal forward reference. A symbol in the statement corresponding to the error location that should have been previously defined is not previously defined.
8	Illegal term. A term has an illegal value less than zero or greater than 15.

The last line of the listing is an error summary as follows:

```
0006 ERS
```

In an error-free listing the statement is printed with four zeros as the number of errors.

Figure 10-1 shows an example of a source listing with errors and an undefined symbol. The error messages shown precede the statement in which the error was detected because these errors were detected as the statements were read. Following the last source line, the undefined symbol is printed. The symbol MULT in the DEF directive is undefined because it does not appear in the label field of a source statement. The error summary line follows the list of undefined symbols.

10.2 OBJECT CODE

The Assembler produces object code that may be linked to other object code modules or programs and loaded into the Model 990 computer, or may be loaded into the computer directly. Object code consists of records containing up to 71 ASCII characters each. The format, described in the next paragraph, permits correction using a keyboard device. Re-assembly to correct errors is unnecessary. An example of output code is included in appendix B.

10.2.1 OBJECT CODE FORMAT

The object record consists of a number of tag characters, each followed by one or two fields as defined in table 10-2. The first character of a record is the first tag character, which tells the loader which field or pair of fields follows the tag. The next tag character follows the end of the field or pair of fields associated with the preceding tag character. When the assembler has no more data for the record, the assembler writes the tag character 7 followed by the check sum field, and the tag character F, which requires no fields. The assembler then fills the rest of the record with blanks, and begins a new record with the appropriate tag character.

Tag character 0 is followed by two fields, and appears at the beginning and end of the object code file. The first field is zero in the first occurrence of tag character 0 and is the number of bytes of relocatable code in the last occurrence. In the first occurrence the second field contains the program identifier assigned to the program by an IDT statement. When no IDT statement is entered, the second field contains blanks. In the last occurrence of the tag character 0, the second field contains blanks. The loader uses the program identifier to identify the program, and the number of bytes of relocatable code to determine the load bias for the next module or program.

Tag characters 1 and 2 are used with entry addresses. Tag character 1 is used when the entry address is absolute. Tag character 2 is used when the entry address is relocatable. The hexadecimal field contains the entry address. One of these tags may appear at the end of the object code file. The

PAGE 0001

```
0001                              IDT   'M1'
0002                              DEF   MULT,MENT,WS
** ERR 5 - LOC 0000 **
0003                              DXOP  MULT,0
0004   0006          WS           BES   6
0005   0006   0001                DATA  1,2,0
       0008   0002
       000A   0000
** ERR 2 - LOC 000C **
0006                              BES   20
0007   000C   C6DB   MENT         MOV   *11,*11
0008   000E   11--                JLT   MCND
0009   0010   C03E                MOV   *14+,0
0010   0012   C050                MOV   *0,1
0011   0014   11--                JLT   MPR
0012   0016   385B                MPY   *11,1
0013   0018   2143                COC   3,5
0014   001A   13--                JEQ   NEG
0015   001C   2144                COC   4,5
0016   001E   13--                JEQ   NEG
0017   0020   C401   COM          MOV   1,*0
0018   0022   05C0                INCT  0
0019   0024   C402                MOV   2,*0
0020   0026   0205                LI    5,0
       0028   0000
0021   002A   0380                RTWP
0022   002C   E143   MCND         SOC   3,5
       000E**110E
0023   002E   0460                B     @MENT+4
       0030   0010'
0024   0032   E144   MPR          SOC   4,5
       0014**110E
0025   0034   0460                B     @MENT+10
       0036   0016'
0026   0038   0502   NEG          NEG   2
       001A**130E
       001E**130C
0027   003A   0541                INV   1
0028   003C   10F1                JMP   COM
0029                              END
MULT
0002 ERS
```

Figure 10-1. Source Listing with Error Messages

Table 10-2. Object Output Tags Supplied by the Assembler

Tag Character	Hexadecimal Field (Four Characters)	Second Field	Meaning
0	Length of all relocatable code	8-character Program Identifier	Program Start
1	Entry address	None	Absolute Entry Address
2	Entry address	None	Relocatable Entry Address
3	Location of last appearance of symbol	6-character symbol	External Reference last used in relocatable code
4	Location of last appearance of symbol	6-character symbol	External Reference last used in absolute code
5	Location	6-character symbol	Relocatable External Definition
6	Location	6-character symbol	Absolute External Definition
7	Checksum for current record	None	Checksum
9	Load address	None	Absolute load address
A	Load address	None	Relocatable load address
B	Data	None	Absolute data
C	Data	None	Relocatable data
F	None	None	End-of-record

associated field is used by the loader to determine the entry point at which execution starts when the loading is complete.

Tag characters 3 and 4 are used for external references. Tag character 3 is used when the last appearance of the symbol in the second field is in relocatable code. Tag character 4 is used when the last appearance of the symbol is absolute code. The hexadecimal field contains the location of the last appearance.

The symbol in the second field is the external reference. Both fields are used by the linking loader to provide the desired linking to the external reference.

For each external reference in a program, there is a tag character in the object code, with a location, or an absolute zero, and the symbol that is referenced. When the object code field contains absolute zero, no location in the program requires the address that corresponds to the reference (an IDT character string, for example). Otherwise, the address corresponding to the reference will be placed in the location specified in the object code by the linking loader. The location specified in the object code similarly contains absolute zero or another location. When it contains absolute zero, no further linking is required. When it contains a location, the address corresponding to the reference will be placed in that address by the linking loader. The location of each appearance of a reference in a program contains either an absolute zero or another location into which the linking loader will place the referenced address.

Figure 10-2 illustrates the chain of the external reference EXTR. The object code contains the following tag and fields:

4C00EEXTR

At location C00E, the address C00A points to the preceding appearance of the reference. The chain includes both absolute and relocatable addresses and consists of absolute addresses C00E, C00A, C006, and C002, relocatable addresses 029E, 029A, and 0298, absolute addresses B00E, B00A, B006, and B002, and relocatable addresses 0290 and 028E. Each location points to the preceding appearance, except for location 028E, which contains zero. The zero identifies location 028E as the first appearance of EXTR, the end of the chain.

Tag characters 5 and 6 are used for external definitions. Tag character 5 is used when the location is relocatable. Tag character 6 is used when the location is absolute. Both fields are used by the linking loader to provide the desired linking to the external definition. The second field contains the symbol of the external definition.

Tag character 7 precedes the checksum, which is an error detection word. The checksum is formed as the record is being written. It is the two's complement of the sum of the 8-bit ASCII values of the characters of the record from the first tag of the record through the checksum tag, 7.

Tag characters 9 and A are used with load addresses for data that follows. Tag character 9 is used when the load address is absolute. Tag character A is used when the load address is relocatable. The hexadecimal field contains the address at which the following data word is to be loaded. A load address is required for a data word that is to be placed in memory at some address other than the next address. The load address is used by the loader.

```
MIRA TEST PROGRAM          990990--9909     **          PAGE 0007

0229                    *
0230                    *       DEMONSTRATE EXTERNAL REFERENCE LINKING
0231                    *
0232                            REF  EXTR
0233  028C                      RORG
0234  028C  C820               MOV  @EXTR,@EXTR
      028E  0000
      0290  028E'
0235  0292  28E0               XOR  @EXTR,3
      0294  0290'
0236  B000                      AORG >B000
0237  B000  3220               LDCR @EXTR,8
      B002  0294'
0238  B004  0420               BLWP @EXTR
      B006  B002
0239  B008  0223               AI   3,EXTR
      B00A  B006
0240  B00C  38A0               MPY  @EXTR,2
      B00E  B00A
0241  0296                      RORG
0242  0296  C820               MOV  @EXTR,@EXTR
      0298  B00E
      029A  0298'
0243  029C  28E0               XOR  @EXTR,3
      029E  029A'
0244  C000                      AORG >C000
0245  C000  3220               LDCR @EXTR,8
      C002  029E'
0246  C004  0420               BLWP @EXTR
      C006  C002
0247  C008  0223               AI   3,EXTR
      C00A  C006
0248  C00C  38A0               MPY  @EXTR,2
      C00E  C00A
```

Figure 10-2. External Reference Example

Tag characters B and C are used with data words. Tag character B is used when the data is absolute; an instruction word or a word that contains text characters or absolute constants, for example. Tag character C is used for a word that contains a relocatable address. The hexadecimal field contains the data word. The loader places the data word in the memory location specified in the preceding load address field, or in the memory location that follows the preceding data word.

Tag character F indicates the end of a record. It may be followed by blanks. The last record of an object code file has a colon (:) in the first character position of the record, followed by blanks.

10.2.2 MACHINE LANGUAGE FORMAT

Some of the data words preceded by tag character B represent machine instructions. Comparing the source listing with the object code fields identifies the data words that represent machine instructions. Figure 10-3 shows the manner in which the bits of the machine instructions relate to the operands in the source statements for each format of machine instructions.

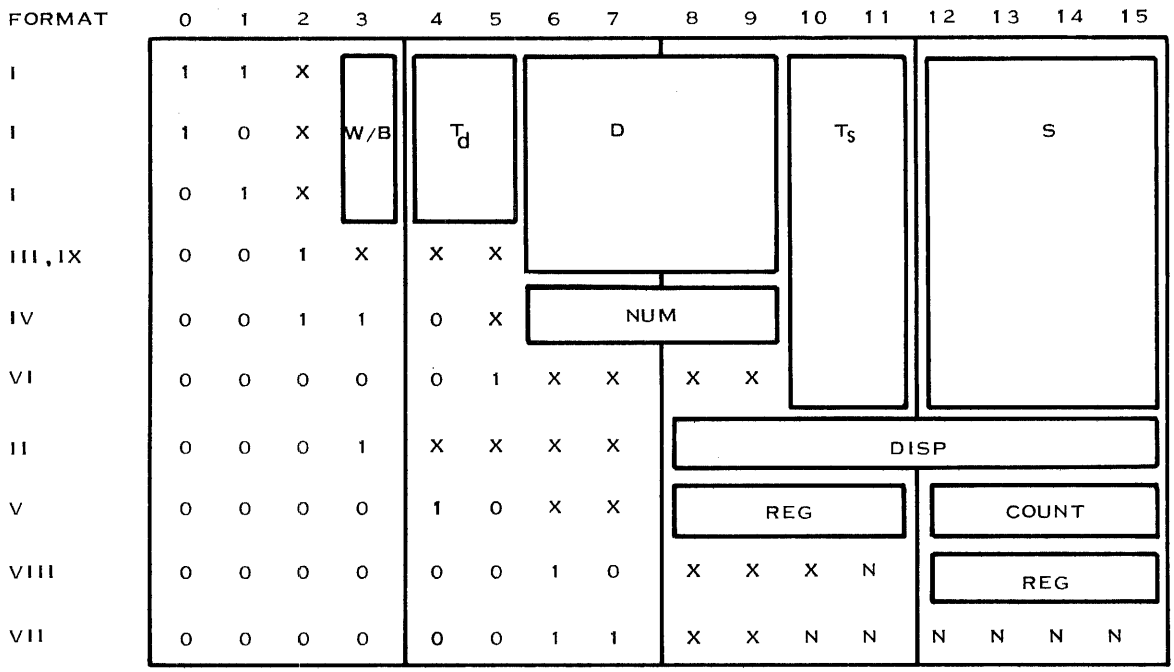
10.3 PROCEDURES FOR CHANGING OBJECT CODE

To correct object code without reassembling a program, change the object code by changing or adding one or more records. One additional tag character is recognized by the loader to permit specifying a load point. The additional tag character, D, may be used in object records changed or added manually.

Tag character D is followed by a load bias (offset) value. The loader uses this value instead of the load bias computed by the loader itself. The loader adds the load bias to all relocatable entry addresses, external references, external definitions, load addresses, and data. The effect of the D tag character is to specify the area of memory into which the loader loads the program.

Correction of object code may require only changing a character or a word in an object code record. The user may duplicate the record up to the character or word in error, replace the incorrect data with the correct data, and duplicate the remainder of the record up to the 7 tag character. Because the changes the user has made will cause a checksum error when the checksum is verified as the record is loaded, the user must change the 7 tag character to F.

When more extensive changes are required, the user may write an additional object code record or records. Begin each record with a tag character 9 or A followed by an absolute load address or a relocatable load address, respectively. This may be an address into which an existing object code record places a different value. The new value on the new record will override the



(A)128442

X is a bit of the operation code that is either 0 or 1 according to the specific instruction in the format

W/B is a bit of the operation code that is 0 in instructions that operate on words, and 1 in instructions that operate on bytes

T_d is a pair of bits that specify the addressing mode of the destination operand, as follows:

00 = Workspace register addressing
01 = Workspace register indirect addressing
10 = Symbolic memory addressing when D = 0
10 = Indexed memory addressing when D ≠ 0
11 = Workspace register indirect autoincrement addressing

D is the workspace register for the destination operand

T_s is a pair of bits that specify the addressing mode of the source operand as shown for T_d

S is the workspace register for the source operand

NUM is the number of bits to be transferred

DISP is a two's complement number that represents a displacement

REG is a workspace register address

COUNT is a shift count

Figure 10-3. Machine Instruction Formats

other value when the new record follows the other record in the loading sequence. Follow the load address with a tag character B or C and an absolute data word or a relocatable data word, respectively. Additional data words preceded by appropriate tag characters may follow. When additional data is to be placed at a non-sequential address, write another load address tag character followed by the load address and data words preceded by tag characters. When the record is full, or all changes have been written, write tag character F to end the record.

When additional memory locations are loaded as a result of changes, the user must change the hexadecimal field following the tag character 0 at the end of the object code file. For example, when the object file written by the assembler contained 1000_{16} bytes of relocatable code, and the user has added 8 bytes in a new object record, additional memory locations will be loaded. The user must find the 0 tag character at the end of the object code file and change the value following the tag character from 1000 to 1008; he must also change the 7 tag character to F in that record.

When added records place corrected data in locations previously loaded, the added records must follow the incorrect records. The loader processes the records as they are read from the object medium, and the last record that affects a given memory location determines the contents of that location at execution time.

The object code records that contain the external definition fields, the external reference fields, the entry address field, and the final program start field must follow all other object records. An additional field or record may be added to include reference to a program identifier. The tag character is 4, and the hexadecimal field contains zeros. The second field contains the first six characters of the IDT character string. External definitions may be added using tag character 5 or 6 followed by the relocatable or absolute address, respectively. The second field contains the defined symbol, filled to the right with blanks when the symbol contains less than six characters.

APPENDIX A
CHARACTER SET

APPENDIX A

CHARACTER SET

The Model 990 Assembly Language uses the ASCII characters listed in table A-1. The table includes the ASCII code for each character, represented as a hexadecimal value and as a decimal value. The table also shows the corresponding Hollerith code. In addition to the characters listed in table A-1, Model 990 Assembly Language defines six characters that are undefined in ASCII. Table A-2 lists these characters, hexadecimal and decimal representations, corresponding Hollerith codes, and the corresponding character on the Model 29 keypunch.

Table A-1. Character Set

Hexadecimal Value	Decimal Value	Character	Hollerith Code
20	32	Space	Blank
21	33	!	11-8-2
22	34	"	8-7
23	35	#	8-3
24	36	$	11-8-3
25	37	%	0-8-4
26	38	&	12
27	39	'	8-5
28	40	(	12-8-5
29	41	)	11-8-5
2A	42	*	11-8-4
2B	43	+	12-8-6
2C	44	,	0-8-3
2D	45	-	11
2E	46	.	12-8-3
2F	47	/	0-1
30	48	0	0
31	49	1	1
32	50	2	2
33	51	3	3
34	52	4	4
35	53	5	5
36	54	6	6
37	55	7	7
38	56	8	8
39	57	9	9
3A	58	:	8-2
3B	59	;	11-8-6

Table A-1. Character Set (Continued)

Hexadecimal Value	Decimal Value	Character	Hollerith Code
3C	60	<	12-8-4
3D	61	=	8-6
3E	62	>	0-8-6
3F	63	?	0-8-7
40	64	@	8-4
41	65	A	12-1
42	66	B	12-2
43	67	C	12-3
44	68	D	12-4
45	69	E	12-5
46	70	F	12-6
47	71	G	12-7
48	72	H	12-8
49	73	I	12-9
4A	74	J	11-1
4B	75	K	11-2
4C	76	L	11-3
4D	77	M	11-4
4E	78	N	11-5
4F	79	O	11-6
50	80	P	11-7
51	81	Q	11-8
52	82	R	11-9
53	83	S	0-2
54	84	T	0-3
55	85	U	0-4
56	86	V	0-5
57	87	W	0-6
58	88	X	0-7
59	89	Y	0-8
5A	90	Z	0-9

Table A-2. Additional Characters

Hexadecimal Value	Decimal Value	Character	Hollerith Code	Keypunch Character
5B	91	[	12-2-8	¢
5C	92	\	0-8-2	0-8-2
5D	93	]	12-7-8	¦ (vertical bar)
5E	94	^	11-7-8	¬ (logical NOT)
5F	95	_	0-5-8	_ (underscore)
00	00	Null		
09	09	Tab		

APPENDIX B
SAMPLE PROGRAM

APPENDIX B

SAMPLE PROGRAM

This appendix describes a sample program in Model 990 Assembly Language, and includes the coding sheets, the source listing, and the contents of the object records.

The program translates 80 ASCII characters from buffer BUFF and places the result in buffer OUT as hexadecimal values. The seven-bit ASCII values are assumed to have been placed in the bytes of BUFF right-justified with leading zeros. The program translates characters 0 through 9 and A through F correctly, but does not check that the characters in BUFF are within that range. When the translation is complete the computer enters the idle mode awaiting an interrupt.

The program consists of a main program and an extended operation subroutine. The main program consists of a loop that is executed for each word in buffer BUFF. The loop makes a correction for characters A through F and masks out the four most significant bits of each byte. The loop then packs the remaining bits of the word into a byte and stores the byte in buffer OUT. The extended operation subroutine provides an AND words operation using the symbol AND to specify the extended operation.

The coding sheets for the sample program are shown in figure B-1. The first statement, a TITL directive, is placed first in order to have the title on the first page of the listing, figure B-2. The IDT directive supplies a program name to the linking loader. The AORG directive provides a block of absolute code to initialize the pair of words at absolute address 40_{16}. The DATA directive places the addresses of the subroutine workspace and the subroutine entry point in this pair of words. The RORG directive causes the remaining code of the program to be relocatable. The TITL directive changes the title on the second sheet of the listing, and the PAGE directive forces a new page of the listing.

The EVEN directive assures that the area reserved for the workspace begins on a word boundary. The BSS directive reserves an area for the subroutine workspace, at location WS. The DXOP directive assigns the symbol AND to extended operation 0. The MOV instruction at location ANDS moves the operand in the workspace of the calling program into workspace register 1 of the subroutine workspace. The next MOV instruction moves the contents of the address in workspace register 11, the other operand, to the location of the immediate value for the ANDI instruction. The next instruction, ANDI, performs the AND operation between the operands and places the result in workspace register 1. The MOV instruction transfers the result into the calling program workspace, and the RTWP instruction returns control to the calling program. The RTWP instruction also restores the calling program environment. Another TITL directive changes the title for the third page of the listing, and a PAGE directive forces a new listing page.

```
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53
01          TITL 'SAMPLE PRØGRAM'
02
03          IDT  'SAMPRØG'
04
05          AØRG >40          LØAD DEDICATED ADDRESSES
06
07          DATA WS,ANDS      FØR XØP 0
08
09          RØRG
10
11          TITL 'AND SUBRØUTINE'
12
13          PAGE
14
15 *        THIS SUBRØUTINE PERFØRMS AN AND ØPERATIØN
16
17 *        BETWEEN THE ØPERAND ØF THE DEFINED XØP, AND, AND
18
19 *        THE CØNTENTS ØF THE CALLING PRØGRAM'S WØRKSPACE
20
21 *        REGISTER 1. THE SUBRØUTINE PLACES THE RESULT IN
22
23 *        THE CALLING PRØGRAM'S WØRKSPACE REGISTER 1. THE
24
25 *        CALLING SEQUENCE IS AS FØLLØWS:
26
27 *
28
29 *            AND  @B       CALL SUBRØUTINE
30
```

(A)128443 (1/4)

Figure B-1. Coding Sheets (Sheet 1 of 4)

```
*
        EVEN                    WØRKSPACE ØN WØRD BØUNDARY
WS      BSS   32                SUBRØUTINE WØRKSPACE

        DXØP  AND,0             DEFINE AND ØPERATIØN
ANDS    MØV   @2(13),1          MØVE CALLING W R 1 TØ W R 1

        MØV   *11,@x+6          MØVE ØPERAND INTØ
        ANDI  1,0               IMMEDIATE INSTRUCTIØN
        MØV   1,@2(13)          MØVE W R 1 TØ CALLING W R 1

        RTWP                    RETURN

        TITL 'MAIN PRØGRAM'

        PAGE
*   THIS PRØGRAM TRANSLATES 80 ASCII CHARACTERS IN
*   BUFF INTØ HEXADECIMAL VALUES. ANY CHARACTER IS
*   TRANSLATED; CHARACTERS 0 - 9 AND A - F ARE
*   TRANSLATED CØRRECTLY. THE 40 WØRDS RESULTING
```

(A)128443 (2/4)

Figure B-1. Coding Sheets (Sheet 2 of 4)

```
01  *     FROM THE TRANSLATION ARE PLACED IN OUT.
02
03  *
04
05            EVEN                  WORKSPACE ON WORD BOUNDARY
06
07  W1        BSS   32              RESERVE MAIN WORKSPACE
08
09  BUFF      BSS   80              INPUT BUFFER
10
11  OUT       BSS   40              OUTPUT BUFFER
12
13  ADDR      DATA  BUFF,OUT        DEFINE ADDRESSES
14
15  START     LWPI  W1              LOAD WORKSPACE POINTER
16
17            LI    0,>0F0F         LOAD W R 0
18
19            MOV   @ADDR,2         LOAD W R 2
20
21            CLR   3               SET W R 3 TO ZERO
22
23            MOV   @ADDR+2,5       LOAD W R 5
24
25            MOV   @START+2,6      LOAD W R 6
26
27  LOOP      MOV   *2+,1           LOAD W R 1
28
29            CI    1,>3A00         JUMP IF
30
```

(A)128443 (3/4)

Figure B-1. Coding Sheets (Sheet 3 of 4)

```
        JLT   SHFT        0 THRØUGH 9
        AI    1,>0900     CØRRECT A THRØUGH F
SHFT    SWPB  1           EXCHANGE CHARACTERS

        INV   3           INVERT W R 3
        JLT   LØØP+2      CHECK ØTHER CHARACTER

        AND   @W1         SUBRØUTINE CALL
        MØV   1,4         STØRE RESULT
        SRC   4,4         SHIFT RESULT

        SØC   4,1         ØR DIGITS
        MØVB  @3(6),*5+   STØRE CHARACTER

        CI    2,BUFF+80   TEST FØR MØRE DATA
        JLT   LØØP        MØRE DATA

        IDLE              AWAIT INTERRUPT

        END   START       FINISH
```

(A)128443 (4/4)

Figure B-1. Coding Sheets (Sheet 4 of 4)

```
SAMPLE PROGRAM                                          PAGE 0001

0002                        IDT  'SAMPROG'
0003  0040                  AORG >40            LOAD DEDICATED ADDRESSES
0004  0040  ----            DATA WS,ANDS        FOR XOP 0
      0042  ----
0005  0000                  RORG
```

Figure B-2. Source Listing (Sheet 1 of 3)

```
AND SUBROUTINE                                          PAGE 0002

0008                    *   THIS SUBROUTINE PERFORMS AN AND OPERATION
0009                    *   BETWEEN THE OPERAND OF THE DEFINED XOP, AND, AND
0010                    *   THE CONTENTS OF THE CALLING PROGRAM'S WORKSPACE
0011                    *   REGISTER 1.  THE SUBROUTINE PLACES THE RESULT IN
0012                    *   THE CALLING PROGRAM'S WORKSPACE REGISTER 1.  THE
0013                    *   CALLING SEQUENCE IS AS FOLLOWS.
0014                    *
0015                    *       AND   @B            CALL SUBROUTINE
0016                    *
0017                          EVEN                  WORKSPACE ON WORD BOUNDARY
0018  0000              WS    BSS   32              SUBROUTINE WORKSPACE
      0040**0000'
0019                          DXOP  AND,0           DEFINE AND OPERATION
0020  0020  C06D        ANDS  MOV   @2(13),1        MOVE CALLING W R 1 TO W R 1
      0022  0002
      0042**0020'
0021  0024  C81B              MOV   *11,@$+6        MOVE OPERAND INTO
      0026  '002A'
0022  0028  0241              ANDI  1,0             IMMEDIATE INSTRUCTION
      002A  0000
0023  002C  CB41              MOV   1,@2(13)        MOVE W R 1 TO CALLING W R 1
      002E  0002
0024  0030  0380              RTWP                  RETURN
```

Figure B-2. Source Listing (Sheet 2 of 3)

```
MAIN PROGRAM                                                PAGE 0003

0027                    *   THIS PROGRAM TRANSLATES 80 ASCII CHARACTERS IN
0028                    *   BUFF INTO HEXADECIMAL VALUES.  ANY CHARACTER IS
0029                    *   TRANSLATED, CHARACTERS 0 - 9 AND A - F ARE
0030                    *   TRANSLATED CORRECTLY.  THE 40 WORDS RESULTING
0031                    *   FROM THE TRANSLATION ARE PLACED IN OUT.
0032                    *
0033                          EVEN              WORKSPACE ON WORD BOUNDARY
0034  0032              W1    BSS  32           RESERVE MAIN WORKSPACE
0035  0052              BUFF  BSS  80           INPUT BUFFER
0036  00A2              OUT   BSS  40           OUTPUT BUFFER
0037  00CA  0052'       ADDR  DATA BUFF,OUT     DEFINE ADDRESSES
      00CC  00A2'
0038  00CE  02E0        START LWPI W1           LOAD WORKSPACE POINTER
      00D0  0032'
0039  00D2  0200              LI   0,>0F0F      LOAD W R 0
      00D4  0F0F
0040  00D6  C040              MOV  @ADDR,2      LOAD W R 2
      00D8  00CA'
0041  00DA  04C3              CLR  3            SET W R 3 TO ZERO
0042  00DC  C160              MOV  @ADDR+2,5    LOAD W R 5
      00DE  00CC'
0043  00E0  C1A0              MOV  @START+2,6   LOAD W R 6
      00E2  00D0'
0044  00E4  C072  LOOP        MOV  *2+,1        LOAD W R 1
0045  00E6  0281              CI   1,>3A00      JUMP IF
      00E8  3A00
0046  00EA  11--              JLT  SHFT         0 THROUGH 9
0047  00EC  0221              AI   1,>0900      CORRECT A THROUGH F
      00EE  0900
0048  00F0  06C1  SHFT        SWPB 1            EXCHANGE CHARACTERS
      00EA**1102
0049  00F2  0543              INV  3            INVERT W R 3
0050  00F4  11F8              JLT  LOOP+2       CHECK OTHER CHARACTER
0051  00F6  2C20              AND  @W1          SUBROUTINE CALL
      00F8  0032'
0052  00FA  C101              MOV  1,4          STORE RESULT
0053  00FC  0B44              SRC  4,4          SHIFT RESULT
0054  00FE  E044              SOC  4,1          OR DIGITS
0055  0100  DD66              MOVB @3(6),*5+    STORE CHARACTER
      0102  0003
0056  0104  0282              CI   2,BUFF+80    TEST FOR MORE DATA
      0106  00A2'
0057  0108  11ED              JLT  LOOP         MORE DATA
0058  010A  0340              IDLE              AWAIT INTERRUPT
0059                          END  START        FINISH
0000 ERS
```

Figure B-2. Source Listing (Sheet 3 of 3)

The main program listing is on page 3. Four directives reserve areas for the workspace and the buffers, and a DATA directive places the buffer addresses in a pair of memory locations.

The first group of instructions beginning at location START initializes the workspace for the processing to follow. The LWPI instruction places the workspace address in the workspace pointer register. The LI instruction places the mask, $0F0F_{16}$, into workspace register 0. The MOV instruction places the address of BUFF into workspace register 2. The CLR instruction clears workspace register 3 to zero to use as a character flag. Another MOV instruction places the address of OUT into workspace register 5. A third MOV instruction places the address of the workspace into workspace register 6.

The processing begins at location LOOP with a MOV instruction that places the first word of BUFF into workspace register 1. The CI instruction tests the leftmost character of the word to determine if it requires modification. The next instruction, JLT, jumps to location SHFT when the leftmost character is 9 or less. Otherwise the AI instruction corrects the character as required for A through F. Then the SWPB instruction at location SHFT exchanges the characters, and the INV instruction inverts the character flag. When only one character of the word has been tested, the character flag is equal to -1 at this point, and the JLT instruction returns control to the CI instruction to process the other character. When both characters have been tested and any necessary correction has been performed, the program calls XOP AND to mask off the most significant four bits of each character. Control returns at the MOV instruction, that transfers the result of the AND operation to workspace register 4. Then the SRC instruction shifts the result four bit positions to the right. The SOC instruction, effectively an OR operation, combines the contents of workspace register 1 and workspace register 4. The rightmost byte of workspace register 1 now contains the hexadecimal values of both characters. The MOVB instruction stores this byte in buffer OUT. The CI instruction determines whether or not all words of BUFF have been converted, and the JLT instruction returns control to location LOOP to process another word until all words have been processed. The IDLE instruction places the computer in the IDLE mode. (The computer remains in the IDLE mode until the operator intervenes or an interrupt occurs. Not all computers have a means of operator intervention. Additional programming not shown in this example is required to properly implement an interrupt.) The last statement is an END directive that causes the assembler to terminate the assembly and supplies the entry location, START, to the loader.

The source listing (figure B-2) consists of a heading on each page, followed by source lines. The heading lines consist of titles supplied by TITL directives, and page numbers. The source lines consist of four columns. The first column contains a statement number. Notice that the TITL and PAGE directives are assigned numbers, and that these numbers do not appear on

the source listing because the directives are not listed. The second column contains the location counter value. This column is blank for directives that do not affect the location counter values. The third column contains the hexadecimal value placed in the location by the assembler. The column is blank for directives that do not provide values. One or more of the hexadecimal digit positions may contain a hyphen (-). This occurs when a forward reference determines the values of these digits. When the statement containing the forward reference has been processed, the assembler prints a line with the location, two asterisks (**), and the complete value. For example, the values shown as hyphens on page 1 are supplied on page 2, following statements 18 and 20, respectively. Notice that some values are followed by an apostrophe ('). The apostrophe indicates that the value is relocatable and will be modified when the program is loaded. The loader modifies relocatable values by adding the load point address to each value. The fourth column contains the source statement supplied to the assembler.

If any errors had been detected by the assembler, the error codes would have been printed as the errors were detected. Following the END statement, the error count (0 in the sample program) is printed.

Figure B-3 shows the contents of the object records assembled for the sample program. Notice that the first tag character (table 9-2) is 0, and that the hexadecimal field is zero, since the assembler has no way of knowing at this point what the length of relocatable code will be. The program identifier follows the hexadecimal field. The next tag character is 9, followed by an absolute address, 0040_{16}, tag character C, and relocatable data, 0000_{16}. This causes the loader to add the load point value to the relocatable value and place the sum in the absolute address. The next tag character, A, is followed by relocatable address 0020_{16}, tag character B, absolute data $C06D_{16}$, tag character B, and absolute data 0002_{16}. This causes the loader to add the load point value to the relocatable address to get the absolute address in which to load the following data. The data words are absolute, and the loader places these words in consecutive address unaltered. The remaining tag characters and fields contain load addresses and data in the order shown in the source listing. Five more tag characters with accompanying hexadecimal fields appear in the first record, followed by tag character 7. The checksum for the first record follows the tag character, and is followed by tag character F, the end-of-record indicator.

The next four records are similar, and contain load addresses and data. Notice that the checksum field and the end-of-record tag that terminate each record do not necessarily appear in the same character positions within the record. The assembler supplies these immediately following the last data field of the record.

The next to the last record begins with tag character 2, followed by relocatable entry address $00CE_{16}$. The loader adds the load point value to the entry

```
00000SAMPROG 90040C0000A0020BC06DB000290042C0020A0024BC81BC002A7F219F
A0028B0241B0000BCB41B0002B0380A00CAC0052C00A2B02E0C0032B0200B0F0F7F1DEF
A00D6BC0A0C00CAB04C3BC160C00CCBC1A0C00D0BC072B0281B3A00A00ECB02217F151F
A00EEB0900B06C1A00EAB1102A00F2B0543B11F8B2C20C0032BC101B0B44BE0447F18EF
A0100BDD66B0003B0282C00A2B11EDB03407F832F
200CE0010C         7FCABF
:
```

Figure B-3. Object Records

address and stores the sum as the address to which control is passed when the load operation is complete. This is followed by tag character 0 and the length of relocatable code, $010C_{16}$. The program identifier field is blank, because the program identifier was supplied in the first record. A checksum field and an end-of-record tag complete the record. The last record consists of the end-of-file indicator, a colon in character position 1.

APPENDIX C
INSTRUCTION TABLES

APPENDIX C
INSTRUCTION TABLES

The source formats for the machine instructions are summarized in eight tables. Refer to the Model 990 Computer Reference Manual for descriptions of the machine instructions. Arithmetic instructions are listed in table C-1, and branch instructions are listed in table C-2. Table C-3 lists compare instructions and table C-4 lists control and CRU instructions. Load and move instructions are listed in table C-5, and logical instructions are listed in table C-6. Workspace register shift instructions are listed in table C-7, and the extended operation instruction is listed in table C-8.

The pseudo-instructions are listed in table C-9.

The following symbols are used in tables C-1 through C-9:

G, G1, G2	- A general address in one of the five modes described in Section III
R	- A workspace register address, described in paragraph 3.2
S	- A symbolic memory address (a label or an expression that contains a label or $)
E	- An expression, described in paragraph 2.2, with the additional limitation that the expression must not contain a symbol that is not previously defined.
I	- An immediate value, which is an expression (paragraph 2.2)
T	- A term, described in paragraph 2.5
(,)	- The contents of the address within parentheses
→	- "replaces"
:	- "is compared to"

The following example shows the use of the symbols in the source format column:

```
XOR   G,R
```

The source format entry means that the mnemonic operation code XOR requires a general address and a workspace register address separated by a comma. In the effect column, the symbols are used as in the following example:

(G) XOR (R) → (R)

This means that the result of an exclusive OR of the contents of the general address with the contents of the workspace register replaces the contents of the workspace register. In the status bits test column, the symbols are used as in the following example:

(R) : 0

This means that the result placed in the workspace register is compared to zero and the status bits contain the result of this comparison.

Table C-1. Arithmetic Instructions

Instruction	Format	Effect	Opcode	Status Bits Affected	Status Bits Test	Format Number
Add words	A G1, G2	(G1)+(G2)→(G2)	A000	0 - 4	(G2) :0	I
Add bytes	AB G1, G2	(G1)+(G2)→(G2)	B000	0 - 5	(G2) :0	I
Absolute value	ABS G	Absolute (G)→(G)	0740	0 - 2	Note 1	VI
Add immediate	AI R, I	(R)+I→(R)	0240	0 - 4	(R) :0	VIII
Decrement	DEC G	(G)-1→(G)	0600	0 - 4	(G) :0	VI
Decrement by 2	DECT G	(G)-2→(G)	0640	0 - 4	(G) :0	VI
Divide	DIV G, R	Note 2	3C00	4	Note 3	IX
Increment	INC G	(G)+1→(G)	0580	0 - 4	(G) :0	VI
Increment by 2	INCT G	(G)+2→(G)	05C0	0 - 4	(G) :0	VI
Multiply	MPY G, R	Note 4	3800	None		IX
Negate	NEG G	-(G)→(G)	0500	0 - 2	(G) :0	VI
Subtract	S G1, G2	(G2)-(G1)→(G2)	6000	0 - 4	(G2) :0	I
Subtract Bytes	SB G1, G2	(G2)-(G1)→(G2)	7000	0 - 5	(G2) :0	I

NOTES

1. The original value of G is compared to zero.
2. The contents of register R and the next consecutive register (32-bit magnitude) are divided by G (16-bit magnitude). The quotient (16-bit magnitude) is placed in R and the remainder is placed in R+1. If R=15, the remainder is placed in the location immediately following the workspace.
3. If the divisor is less than or equal to the left half of the dividend, the divide instruction is aborted and overflow status bit (bit 4) is set.
4. (G) is multiplied by (R). The result (32-bit magnitude) is placed in R and R+1. R contains the most significant half of the result. If R=15, the least significant half of the result if placed in the location immediately following the workspace.

Table C-2. Branch Instructions

Instruction	Format	Effect	Necessary Status	Opcode	Format Number
Branch	B G	G→(PC)	Unconditional	0440	VI
Branch and Link	BL G	G→(PC) (PC→(R11)	Unconditional	0680	VI
Branch and Link WP	BLWP G	Note 1	Unconditional	0400	VI
Jump If Equal	JEQ S	S→(PC)	Bit 2 = 1	1300	II
Jump If High or Equal	JHE S	S→(PC)	Bit 0 or Bit 2 = 1	1400	II
Jump If Greater Than	JGT S	S→(PC)	Bit 1 = 1	1500	II
Jump If Logical High	JH S	S→(PC)	Bit 0 = 1 and Bit 2 = 0	1B00	II
Jump If Logical Low	JL S	S→(PC)	Bit 0 = 0 and Bit 2 = 0	1A00	II
Jump If Less or Equal	JLE S	S→(PC)	Bit 1 = 0 or Bit 2 = 1	1200	II
Jump If Less Than	JLT S	S→(PC)	Bit 1 = 0 and Bit 2 = 0	1100	II
Unconditional Jump	JMP S	S→(PC)	Unconditional	1000	II
Jump If No Carry	JNC S	S→(PC)	Bit 3 = 0	1700	II
Jump If Not Equal	JNE S	S→(PC)	Bit 2 = 0	1600	II
Jump If No Overflow	JNO S	S→(PC)	Bit 4 = 0	1900	II
Jump If Odd Parity	JOP S	S→(PC)	Bit 5 = 1	1C00	II
Jump On Carry	JOC S	S→(PC)	Bit 3 = 1	1800	II

Table C-2. Branch Instructions (Continued)

Instruction	Format	Effect	Necessary Status	Opcode	Format Number
Return WP	RTWP	Note 2	Unconditional	0380	VII
Execute	X G	Note 3	Unconditional	0480	VI

NOTES

1. BLWP is explained in detail in paragraph 8.2. It can be summarized as follows:

   (G)⟶(WP)
   (G + 2)⟶(PC)
   (original WP)⟶(R13)
   (old PC)⟶(R14)
   (ST)⟶(R15)

2. RTWP is explained in detail in paragraph 8.2. It can be summarized as follows:

   (R13)⟶(WP)
   (R14)⟶(PC)
   (R15)⟶(ST)

3. An instruction at address G is executed as if it were located in memory where the Execute instruction resides. Observe that if the instruction executed is not a single word instruction, the word following the Execute instruction is used (i.e., if symbolic memory addressing or indexed addressing is required, the symbol value must be in the word following the Execute instruction). The Execute instruction does not affect the status bits but the instruction executed will set the status bits appropriately.

Table C-3. Compare Instructions

Instruction	Format	Opcode	Status Bits Affected	Status Bits Test	Format Number
Compare Words	C G1, G2	8000	0 - 2	(G1) :(G2)	I
Compare Bytes	CB G1, G2	9000	0 - 2, 5	(G1) :(G2)	I
Compare Immediate	CI R, I	0280	0 - 2	(R) :I	VIII
Compare Ones Corresponding	COC G, R	2000	2	Note 1	III
Compare Zeros Corresponding	CZC G, R	2400	2	Note 2	III

NOTES

General: Compare instructions have no effect other than setting status bits. Note that in two's complement representation negative numbers are logically greater than positive numbers, and that negative numbers of small magnitude are logically greater than negative numbers of larger magnitude.

1. The bits in the destination operand that correspond to bits equal to one in the source operand are compared to one. If the corresponding bits are equal to one, status bit 2 is set to 1. Otherwise the status bit is set to 0.

2. The bits in the destination operand that correspond to bits equal to one in the source operand are compared to zero. If the corresponding bits are equal to zero, status bit 2 is set to 1. Otherwise the status bit is set to 0.

Table C-4. Control and CRU Instructions

Instruction	Format	Effect	Opcode	Status Bits Affected	Status Bits Test	Format Number
Clock Off	CKOF	Note 1	03C0	None		VII
Clock On	CKON	Note 2	03A0	None		VII
Load Communication Register	LDCR G, T	Note 3	3000	0 - 2, 5	(G) :0	IV
Idle	IDLE	Note 4	0340	None		VII
Reset I/O	RSET	Note 5	0360	0 - 5	Note 6	VII
Set Bit to One	SBO E	Note 7	1D00	None		II
Set bit to Zero	SBZ E	Note 8	1E00	None		II
Store Communication Register	STCR G, T	Note 9	3400	0 - 2, 5	(G) :0	IV
Test Bit	TB E		1F00	2	Note 10	II

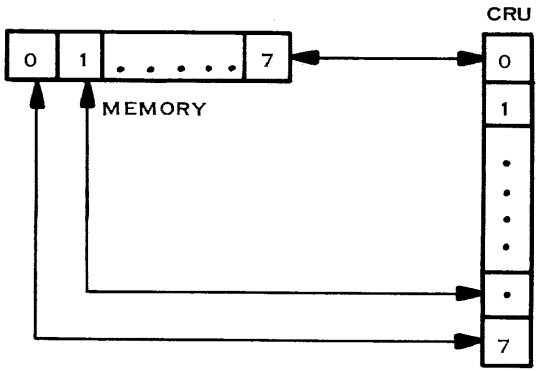
NOTES

1. Disables 120 HZ clock.
2. Enables 120 HZ clock. If interrupt level 5 is enabled, an interrupt occurs every 8.33 ms. Interrupt address is 14_{16}.
3. Transfers consecutive data bits from the byte address specified by G to the CRU. The number of bits transferred is specified by T. The CRU address is the contents of R12 of the current workspace. The least significant bit of the byte addressed by G is placed in the CRU bit addressed by R12. See illustration, Memory CRU Transfer (Note 9).
4. Places the computer in the idle state. An interrupt or start signal causes the computer to resume execution at the instruction following the IDLE instruction.
5. Disables all interrupts. Resets all directly connected I/O devices.

Table C-4. Control and CRU Instructions (Continued)

NOTES

6. Sets bits 0 - 5 to zero.
7. Sets CRU bit at address in R12 + E to one.
8. Sets CRU bit at address in R12 + E to zero.
9. Transfers consecutive data bits from the CRU to the byte address specified by G. The number of bits transferred is specified by T. The CRU address is the contents of R12 of the current workspace. The CRU bit addressed by R12 is placed in the least significant bit of the byte addressed by G. See Memory - CRU Transfer illustration.
10. Tests CRU bit at address in R12 + E. Set status bit 2 to the value of the CRU bit.



(A)128444

Memory - CRU Transfer

Table C-5. Load and Move Instructions

Instruction	Format	Effect	Opcode	Status Bits Affected	Status Bits Test	Format Number
Load Immediate	LI R,I	I→(R)	0200	None		VIII
Load Interrupt Mask	LIMI I	Note 1	0300	None		VIII
Load from ROM and Execute	LREX	Note 2	03E0	None		VII
Load Workspace Pointer	LWPI I	I→(WP)	02E0	None		VIII
Move Words	MOV G1, G2	(G1)→(G2)	C000	0 - 2	(G2) :0	I
Move Bytes	MOVB G1, G2	(G1)→(G2)	D000	0 - 2, 5	(G2) :0	I
Store Status	STST R	(ST)→(R)	02C0	None		VIII
Store WP	STWP R	(WP)→(R)	02A0	None		VIII
Swap Bytes	SWPB G	Note 3	06C0	None		VI

NOTES

1. Places the least-significant 4 bits of the immediate value I in the interrupt mask.
2. Loads the 256 words of the ROM program into the first 256 words of memory. Places the contents of the memory pair at address 0 into WP and PC and starts execution
3. Interchanges bits 0 - 7 with bits 8 - 15 of word at address specified by G.

Table C-6. Logical Instructions

Instruction	Format	Effect	Opcode	Status Bits Affected	Status Bits Test	Format Number
AND Immediate	ANDI R, I	(R) AND I→(R)	0240	0 - 2	(R) :0	VIII
Clear	CLR G	0→(G)	04C0	None		VI
Invert Bits	INV G	Note 1	0540	0 - 2	(G) :0	VI
OR Immediate	ORI R, I	(R) OR I→(R)	0260	0 - 2	(R) :0	VIII
Set to Ones	SETO G	>FFFF→(G)	0700	None		VI
Set Ones Corresponding	SOC G1, G2	Note 2	E000	0 - 2	(G2) :0	I
Set Ones Corresponding Bytes	SOCB G1, G2	Note 2	F000	0 - 2, 5	(G2) :0	I
Set Zeros Corresponding	SZC G1, G2	Note 3	4000	0 - 2	(G2) :0	I
Set Zeros Corresponding Bytes	SZCB G1, G2	Note 3	5000	0 - 2, 5	(G2) :0	I
Exclusive OR	XOR G, R	(G) XOR (R)→(R)	2800	0 - 2	(R) :0	III

NOTES

1. Places one's complement of contents of location G in location G.

2. Sets bits to one in G2 that correspond to bits equal to one in G1. (G1) OR (G2) →(G2).

```
1111111100000000 G1
1010101010101010 G2
1111111110101010 G2 (result)
```

3. Sets bits to zero in G2 that correspond to bits equal to one in G1. (INV(G1)) AND (G2)→(G2).

```
1111111100000000 G1
1010101010101010 G2
0000000010101010 G2 (result)
```

Table C-7. Workspace Register Shift Instructions

Instruction	Format	Value Placed in Vacated Bit Position on Each Shift	Opcode	Format Number
Shift Right Arithmetic	SRA R,C	Original value of leftmost bit	0800	V
Shift Right Logical	SRL R,C	Logical zero	0900	V
Shift Left	SLA R,C	Logical zero (Note 1)	0A00	V
Shift Right	SRC R,C	Rightmost bit moves to leftmost bit	0B00	V

NOTES

General: If C is zero, the 4 least-significant bits of R0 contain the shift value. If the 4 least-significant bits of R0 equal 0, shift 16 positions. Otherwise, shift C positions. The value of the last bit shifted out of the register is placed in status bit 3. The shifted value is compared to zero-setting status bits 0 - 2.

1. If the sign of the value in R changes during shift, sets status bit 4.

Table C-8. Extended Operation Instruction

Instruction	Format	Effect	Opcode	Status Bits Affected	Status Bits Test	Format Number
Extended Operation	XOP G, T	Note 1	2C00	6	Note 2	IX

NOTES

1. T specifies the extended operation, 0 - 15, to be executed. (Paragraph 6.11).
2. Sets status bit 6 to one when extended operation is software implemented, and to zero when extended operation is hardware implemented.

Table C-9. Pseudo-Instructions

Instruction	Equivalent Instruction	Opcode
NOP	JMP $ + 2	1000
RT	B *11	045B

APPENDIX D
ASSEMBLER DIRECTIVE TABLE

APPENDIX D

ASSEMBLER DIRECTIVE TABLE

The assembler directives for the Model 990 Assembly Language are listed in table D-1. All directives may include a comment field following the operand field. Those directives that do not require an operand field may have a comment field following the operator field. Those directives that have optional operand fields (RORG and END) may have comment fields only when they have operand fields.

The following symbols and conventions are used in defining the syntax of assembler directives:

- Angle brackets (< >) enclose items supplied by the user
- Brackets ([ ]) enclose optional items
- An ellipsis (...) indicates that the preceding item may be repeated

The following words are used in defining the items used in assembler directives:

- symbol - defined in paragraph 2.4
- label - a symbol used in the label field
- string - a character string defined in paragraph 2.6, of a length defined for each directive
- expr - an expression, defined in paragraph 2.2.1
- wd expr - well-defined expression defined in paragraph 2.2.2
- term - defined in paragraph 2.5

Table D-1. Assembler Directives

Directive	Syntax	Force Word Boundary	Note
Page Title	[<label>] TITL <string>	NA	
Program Identifier	[<label>] IDT <string>	NA	
External Definition	[<label>] DEF <symbol> [,<symbol>] ...	NA	
External Reference	[<label>] REF <symbol> [,<symbol>] ...	NA	
Absolute Origin	[<label>] AORG <wd expr>	No	
Relocatable Origin	[<label>] RORG [<expr>]	No	1, 3
Block Starting with Symbol	<label> BSS <wd expr>	No	
Block Ending with Symbol	<label> BES <wd expr>	No	
Initialize Word	[<label>] DATA <expr> [,<expr>] ...	Yes	
Initialize Text	[<label>] TEXT [-] <string>	No	2
Define Extended Operation	[<label>] DXOP <symbol>, <term>	NA	
Define Assembly-Time Constant	<label> EQU <expr>	NA	3
Word Boundary	[<label>] EVEN	Yes	
No Source List	[<label>] UNL	NA	
List Source	[<label>] LIST	NA	
Page Eject	[<label>] PAGE	NA	
Initialize Byte	[<label>] BYTE <wd expr> [,<wd expr>] ...	No	
Program End	[<label>] END [<symbol>]	NA	4

NOTES

1. The expression must be relocatable.
2. The minus sign causes the assembler to negate the right-most character.
3. Symbols in expressions must have been previously defined.
4. Symbol must have been previously defined.

USER'S RESPONSE SHEET

Model 990 Computer
Manual Title: Assembly Language Programmer's Guide (943441-9701)

Date of Manual: 1 June 1974 Date of This Letter: _____

User: _____ Office/Dept. No.: _____

Company: _____

Street Address: _____

City/State/Zip: _____

Please list any discrepancy found in this manual by page, paragraph, figure, or table number in the following space. If there are any other suggestions that you wish to make, feel free to include them. Thank you.

Location in Manual	Comment/Suggestion
_____	_____
_____	_____
_____	_____

CUT ALONG LINE

NO POSTAGE NECESSARY IF MAILED IN U.S.A.
FOLD ON TWO LINES (LOCATED ON REVERSE SIDE), STAPLE AND MAIL

First Class
PERMIT NO. 3135
Austin, Texas

BUSINESS REPLY MAIL
No Postage Necessary if Mailed in the United States

Postage Will Be Paid by

TEXAS INSTRUMENTS INCORPORATED
DIGITAL SYSTEMS DIVISION

P.O. BOX 2909 · AUSTIN, TEXAS 78767

Attn: TECHNICAL PUBLICATIONS, MS 2146

Sales and Service Offices of Texas Instruments are located throughout the United States and in major countries overseas. Contact the Digital Systems Division, Texas Instruments Incorporated, P.O. Box 1444, Houston, Texas 77001, or call (713) 494-5115, for the location of the office nearest to you.

TEXAS INSTRUMENTS
INCORPORATED