MACRO-8
PROGRAMMING
MANUAL

PDP-8

DIGITAL EQUIPMENT CORPORATION • MAYNARD, MASSACHUSETTS

PDP-8 PROGRAMMING MANUAL
MACRO-8

For additional copies order No. DEC-08-CMAA-D from Program Library, Digital Equipment Corporation, Maynard, Mass. Price $2.00

DIGITAL EQUIPMENT CORPORATION • MAYNARD, MASSACHUSETTS

1st Printing October 1965
2nd Printing Revised November 1966
3rd Printing October 1967
4th Printing February 1969

Instruction times, operating speeds and the like are included in this manual for reference only; they are not to be taken as specifications.

PREFACE

The PDP-8 comes to the user complete with an extensive selection of system programs and routines making the full data processing capability of the new computer immediately available to each user, eliminating many commonly experienced initial programming delays.

The PDP-8 programming system takes advantage of the many man-years of program development and field testing by PDP-5 users. Although in many cases PDP-8 programs originated as PDP-5 programs, all utility and functional program documentation is issued in a new, recursive format introduced with the PDP-8. Programs written by users of either the PDP-5 or the PDP-8 and submitted to the users' library (DECUS - Digital Equipment Corporation Users' Society) are immediately available to PDP-8 users. Consequently, users of either computer can take immediate advantage of the continuing program developments for the other.

The MACRO-8 Manual is divided into two parts: Basic Information and the MACRO Language. Part 1 is especially for the new user, while those experienced with assembly programs will want to start with Part 2.

CONTENTS

PART 1

BASIC INFORMATION

PART 2

THE MACRO LANGUAGE

CONTENTS (continued)

ILLUSTRATIONS

TABLES

PART 1

BASIC INFORMATION

CHAPTER 1

INTRODUCTION

In describing the solution of the following equation,

$$y=mx+b$$

one can say, "First, multiply the quantity x by m. To this product, add the quantity b. The result is the value of y."

The same problem can be solved on an adding machine in the following steps:

1. Clear the keyboard and registers.

2. Enter the value of m and press the ADD key.

3. Enter the value of x and press the keys which initiate the multiplication.

4. Enter the value of b and press the ADD key. The result, appearing in the totalizing register, is the value of y.

This sequence of steps can be thought of as a program for the solution of the problem on an adding machine. In similar fashion, the steps can be written out for a solution to be performed by a digital computer. Instead of pressing buttons and keys, the programmer writes a sequence of instructions to perform operations on data stored in the computer. Such a sequence (for a hypothetical computer) might appear as follows:

```
clear
add m
multiply x
add b
deposit y
```

The variables m, x, and b represent quantities stored in the computer; the variable y represents a storage register. The operations are carried out in a register called the accumulator, abbreviated AC. The first instruction clears the AC. The next adds the quantity m into the AC. The third instruction multiplies the contents of the AC by the quantity x, and the fourth adds the quantity b to the result. Finally, the contents of the AC are deposited into a storage register designated by the symbol y.

To be useful to a computer, the instructions of a written program must be translated into a sequence of numeric codes, each of which represents a specific computer operation. To do such translating by hand

from instructions to binary numbers would be tedious and lengthy. Computer programs have been written to perform the translating task, interpreting the written program and producing from it a second program which can be executed directly by the computer. These translators are called assemblers because they assemble from a source program written by the user, a working program instruction by instruction. This output from the assembler is called an object, or binary program.

MACRO-8 is an assembler designed to accept input in the form of a sequence of symbolic instructions representing the operations capable of being executed on the PDP-8. It produces a binary program tape which may then be placed in the computer and executed. The next chapter explains a few basics of the binary numbering system; succeeding chapters deal in more detail with the MACRO-8 assembly program.

CHAPTER 2

THE BINARY SYSTEM

Every item of information stored in or processed by a digital computer is encoded as a binary number. Consequently, the user should become familiar with the binary system and be able to convert numbers from binary to decimal representation and back, using the octal system as an intermediate. Eventually, as the programmer gains more experience, he will find himself using the decimal system less and "thinking in octal" more. This is a useful habit to cultivate.

Table 2-1 gives the first 16 integers (and 0) as they are represented in the decimal, binary, and octal number systems. Note that in the decimal system there are ten different symbols, or digits, 0-9 which are used to represent any number. In the binary system there are two, 0 and 1; in the octal system, eight, 0-7. The number of digits required in a given system is called the radix. Therefore, the decimal radix is 10; the octal radix is 8, and the binary radix is 2. A subscript is used to identify the radix of the system in which the number is represented. Thus, 4196_{10} indicates a decimal number, 2547_8 an octal number, and 110101_2 a binary number.

TABLE 2-1 FIRST SIXTEEN INTEGERS IN THREE NUMBER SYSTEMS

Decimal	Octal	Binary	Decimal	Octal	Binary
0	0	000	9	11	1001
1	1	001	10	12	1010
2	2	010	11	13	1011
3	3	011	12	14	1100
4	4	100	13	15	1101
5	5	101	14	16	1110
6	6	110	15	17	1111
7	7	111	16	20	10000
8	10	1000			

All numbering systems using radices involve positional notation; that is, each successive digit position to the left represents the next higher power of the radix. For example, the decimal number 4196_{10} may be expressed algebraically as

$$4 \times 10^3 + 1 \times 10^2 + 9 \times 10^1 + 6 \times 10^0$$

which when calculated becomes

$$4 \times 1000 + 1 \times 100 + 9 \times 10 + 6 \times 1 = 4196_{10}.$$

Positionally, this appears as

10^3	10^2	10^1	10^0	(Positioned radix)
1000	100	10	1	(Radix to its respective powers)
4	1	9	6	(Units required of each value)

Likewise, the octal number 2547_8 can be expressed as $2 \times 8^3 + 5 \times 8^2 + 4 \times 8^1 + 7 \times 8^0$.

In all systems the integral and fractional parts of a number are separated by a radix point. Depending on the system in use, this may be a decimal point, octal point, or binary point.

As can be seen from the table, four binary digit positions are required to represent the decimal integers up to 9. The octal integers up to 7 require only three binary positions; furthermore, exactly three positions are needed. In other words, three binary digit positions are necessary and sufficient to represent the eight octal digits. This fact makes binary-to-octal and octal-to-binary conversions quite simple.

Example: Binary-to-Octal Conversion

The binary integer

$$101110011010_2$$

can be converted to its octal equivalent as follows:

1. Divide the binary digits into groups of three, starting from the right

   101 110 011 010

2. Substitute for each group its octal equivalent

   5 6 3 2

3. The result is the octal number

   5632_8

To perform the reverse conversion, substitute the binary equivalent of each octal digit

$$\begin{matrix} & 4 & 7 & 5 & 1 \\ 4751_8 = & 100 & 111 & 101 & 001 \end{matrix} = 100111101001_2$$

To convert an octal fraction, group by threes in each direction away from the radix point

$$1110101.010001_2 = 1\ 110\ 101\ .\ 010\ 01(0) = 165.22_8$$

$$2243.557_8 = 010\ 010\ 100\ 011\ .\ 101\ 101\ 111 = 010010100011.101101111_2$$

Example: Decimal-to-Binary Conversion

The "remainder method" may be used to convert a decimal number to a binary number. The decimal number

$$2970_{10}$$

may be converted as follows:

	1	1
	$2\overline{)2}$	0
	$2\overline{)5}$	1
	$2\overline{)11}$	1
	$2\overline{)23}$	1
	$2\overline{)46}$	0
	$2\overline{)92}$	0
	$2\overline{)185}$	1
	$2\overline{)371}$	1
	$2\overline{)742}$	0
	$2\overline{)1485}$	1
(binary radix) →	$2\overline{)2970}_{10}$	0 ← (remainders)
		101110011010_2

Notice that this method simply involves halving the number to be converted and noting the remainder after each division.

Example: Decimal-to-Octal Conversion

The remainder method may also be used for octal conversions. The decimal number

$$2970_{10}$$

would be converted as follows:

	0	
	$8\overline{)5}$	5
	$8\overline{)46}$	6
	$8\overline{)371}$	3
(octal radix) →	$8\overline{)2970}_{10}$	2 ← (remainders)
		5632_8

In one sense, the conversion of a number from one representation to another is a way of encoding the number; the octal integer 77_8 can be encoded as the binary integer 111111. Similarly, one can assign a binary code to any symbol, such as a letter of the alphabet. The table in Appendix B shows the binary codes assigned to all the characters of the Model ASR-33 Teletypewriter.

A programmer may invent a symbolic name to refer to the location of a given word in the computer memory. These symbolic names, or tags, are assembled together with the instruction mnemonic into a binary number which indicates the memory location of the word, the instruction code, and the address of the data.

The association of binary code and symbol is the basis of a programming language. A programmer learns the symbols for the computer's repertoire of operations and the rules for arranging a sequence of symbolic instructions in a useful format. He prepares a symbolic program for input on a medium such as punched paper tape. An assembly program accepts this source program input and translates it into an equivalent sequence of binary numbers, producing a program which is usable directly by the computer by placing it on an output medium, which again may be punched tape. This binary, or object program tape may then be read into the computer and executed.

CHAPTER 3

THE PDP-8 INSTRUCTION SET

INSTRUCTIONS

Every PDP-8 operation is specified by a unique combination of 1's and 0's stored in the twelve bits of one memory register. Such an instruction word can be one of two types: memory reference instructions perform operations which require access to the information stored in a memory register; augmented instructions do not refer to memory cells.

The operation code of an instruction is contained in bits 0, 1, and 2 of the word. Since three binary digits correspond to one octal digit, it is apparent that there can be no more than eight operation codes, corresponding to the octal digits 0-7. Codes 0-5 are reserved for memory reference instructions. Operation codes 6 and 7 are for augmented instructions. These two types of instructions are defined, and the instructions described, in the following sections.

The following special symbols are used in the instruction lists below.

Symbol	Definition
C(A)	The contents of register A
C(A) ⇒ C(B)	The contents of register A replace the contents of register B
Y	The address or location of any memory register
Y_j	The jth bit of register Y
Y_{1-4}	Bits 1-4, inclusive, of register Y
$C(A_{0-5}) \Rightarrow C(Y_{6-11})$	The contents of bits 0-5 of register A replace the contents of bits 6-11 of register Y. The contents of A are not affected.
V	Inclusive OR
∧	AND (Boolean)
$\overline{C(A)}$	The 1's complement of the contents of register A

Memory Reference Instructions

Word format of memory reference instructions is shown in Figure 3-1, and the instructions are explained in Table 3-1.

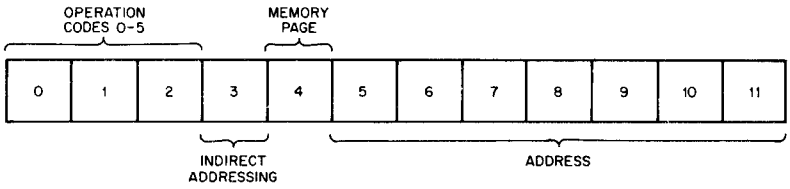

Figure 3-1 Memory Reference Instruction Format

TABLE 3-1 MEMORY REFERENCE INSTRUCTIONS

Mnemonic Symbol	Octal Code	Time (μsec)	Operation
AND Y	0	3.0	Logical AND. The AND operation is performed between the C(Y) and the C(AC). The result is left in the AC, and the original C(AC) are lost. The C(Y) are unchanged. Corresponding bits are compared independently. This instruction, often called extract or mask, can be considered as a bit-by-bit multiplication. $C(Y_i) \wedge C(AC_i) \Rightarrow C(AC_i)$
TAD Y	1	3.0	Two's complement add. The C(Y) are added to the C(AC) in 2's complement arithmetic. The result is left in the AC and the original C(AC) are lost. The C(Y) are unchanged. If there is a carry from AC_0, the link is complemented. This feature is useful in multiple precision arithmetic. C(Y) + C(AC) => C(AC)
ISZ Y	2	3.0	Index and skip if zero. The C(Y) are incremented by one in 2's complement arithmetic. If the resultant C(Y) = 0, the next instruction is skipped. If the resultant C(Y) ≠ 0, the program proceeds to the next instruction. The C(AC) are unaffected. C(Y) + 1 => C(Y) If result = 0, C(PC) + 1 => C(PC)
DCA Y	3	3.0	Deposit and clear AC. The C(AC) are deposited in core memory location Y and the AC is then cleared. The previous C(Y) are lost. C(AC) => C(Y), then 0 => C(AC)

Example

$C(AC_i)$ original	$C(Y_i)$	$C(AC_i)$ final
0	0	0
0	1	0
1	0	0
1	1	1

TABLE 3-1 MEMORY REFERENCE INSTRUCTIONS (continued)

Mnemonic Symbol	Octal Code	Time (μsec)	Operation
JMS Y	4	3.0	Jump to subroutine. The C(PC) are deposited in core memory location Y. The next instruction is taken from location Y + 1. C(PC) + 1 ⇒ C(Y) Y + 1 ⇒ C(PC)
JMP Y	5	1.5	Jump to Y. The next instruction is taken from core memory location Y. Y ⇒ C(PC)

Augmented Instructions

There are two classes of augmented instructions, or instructions which do not reference core memory. They are the input-output transfer (IOT) which has an operation code of 6, and the operate (OPR), which has an operation code of 7. Bits 3 through 11 within these instructions function as an extension of the operation code and can be microprogrammed to perform several operations with one instruction. Augmented instructions are 1-cycle instructions requiring 1.5 μsec for execution.

Input-Output Transfer Instruction

Microinstructions of the input-output transfer (IOT) instruction effect information transfers between the arithmetic and control element and an input-output device via the input-output control element. The format of the IOT instruction is shown in Figure 3-2. Bits 3 through 8 are used to select the I/O device; and bits 9 through 11 enable generation of I/O pulses during event times 4, 2, and 1, respectively. Operations performed by IOT microinstructions are explained in Chapter 4 of the PDP-8 Users Handbook.

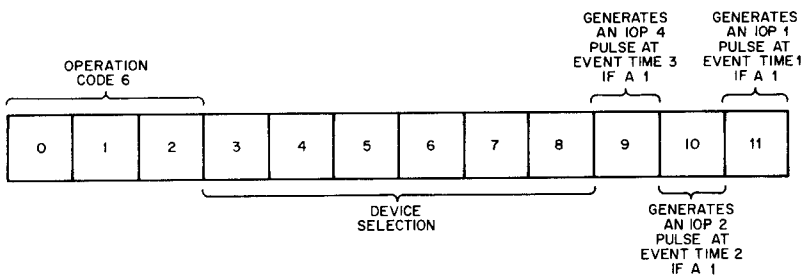

Figure 3-2 IOT Instruction Format

Operate Instruction

The operate instruction consists of two groups of microinstructions. Group 1 is principally for clear, complement, rotate, and increment operations and is designated by the presence of a 0 in bit 3. Group 2 is used principally for checking the contents of the accumulator and link and continuing to or skipping the next instruction based on the check. A 1 in bit 3 and a 0 in bit 11 designate a Group 2 microinstruction.

Group 1 operate microinstruction format is shown in Figure 3-3, and the microinstructions are listed in the table below. Any logical combination of bits within this group can be combined into one microinstruction. For example, it is possible to assign 1's to bits 5, 6, and 11; but it is not logical to assign 1's to bit 8 and 9 simultaneously since they specify conflicting operations. If RAL, RAR, RTL, or RTR is specified, IAC may not be specified, and conversely.

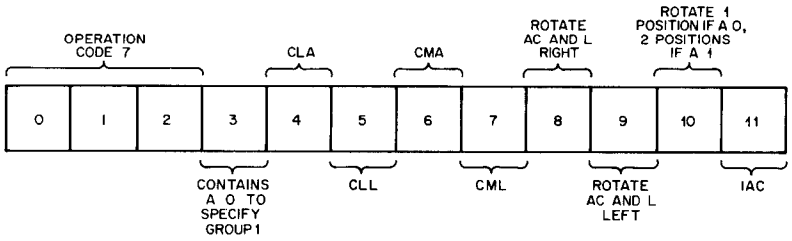

Figure 3-3 Group 1 Operate Microinstruction Format

TABLE 3-2 GROUP 1 OPERATE MICROINSTRUCTIONS

Mnemonic Symbol	Octal Code	Event Time	Operation
CLA	7200	1	Clear AC. 0 ⇒ C(AC)
CLL	7100	1	Clear L. 0 ⇒ C(L)
CMA	7040	1	Complement AC. The C(AC) are set to the 1's complement of C(AC). $\overline{C(AC)} \Rightarrow C(AC)$
CML	7020	1	Complement L. $C(\overline{L}) \Rightarrow C(L)$
RAR	7010	2	Rotate AC and L right. The C(AC) and the C(L) are rotated right one place. $C(AC_i) \Rightarrow C(AC_{i+1})$ $C(AC_{11}) \Rightarrow C(L)$ $C(L) \Rightarrow C(AC_0)$

TABLE 3-2 GROUP 1 OPERATE MICROINSTRUCTIONS (continued)

Mnemonic Symbol	Octal Code	Event Time	Operation
RAL	7004	2	Rotate AC and L left. The C(AC) and the C(L) are rotated left one place. $C(AC_i) \Rightarrow C(AC_{i-1})$ $C(AC_0) \Rightarrow C(L)$ $C(L) \Rightarrow C(AC_{11})$
RTR	7012	2	Rotate two places to the right. Equivalent to two successive RAR operations.
RTL	7006	2	Rotate two places to the left. Equivalent to two successive RAL operations.
IAC	7001	2	Index AC. The C(AC) are incremented by one in 2's complement arithmetic. $C(AC) + 1 \Rightarrow C(AC)$
NOP	7000	—	No operation. Causes a 1.5 μsec program delay.

Group 2 operate microinstruction format is shown in Figure 3-4, and the microinstructions are listed in the table below. Any logical combination of bits within this group can be composed in one microinstruction.

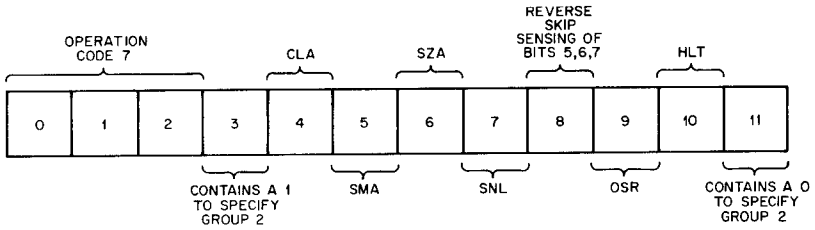

Figure 3-4 Group 2 Operate Microinstruction Format

If skips are combined in a single instruction, the inclusive OR of the conditions determines the skip. For example, if 1's are designated in bits 6 and 7 (SZA and SNL), the next instruction is skipped if either C(AC) = 0, or C(L) = 1, or both. If two reverse sense skip instructions are combined (bit 8 is set), the logical AND of the conditions determines the skip. For example, if 1's are designated in bits 6, 7, and 8 (SNA and SZL), the next instruction is skipped if C(AC) $\neq$ 0 and C(L) = 0. The CLA microinstruction from Group 1 can be combined with Group 2 commands. This command occurs at event time 2 with respect to the event times listed in the following table.

TABLE 3-3 GROUP 2 OPERATE MICROINSTRUCTIONS

Mnemonic Symbol	Octal Code	Event Time	Operation
CLA	7600	2	Clear AC. 0 ⇒ C(AC)
SPA	7510	1	Skip on positive AC. If the C(AC) is a positive number, the next instruction is skipped. If $C(AC_0) = 0$, then C(PC) + 1 ⇒ C(PC)
SMA	7500	1	Skip on minus AC. If the C(AC) is a negative number, the next instruction is skipped. If $C(AC_0) = 1$, then C(PC) + 1 ⇒ C(PC)
SNA	7450	1	Skip on non-zero AC. If C(AC) ≠ 0, then C(PC) + 1 ⇒ C(PC)
SZA	7440	1	Skip on zero AC. If C(AC) = 0, then C(PC) + 1 ⇒ C(PC)
SZL	7430	1	Skip on zero L. If C(L) = 0, the next instruction is skipped. If C(L) = 0, the C(PC) + 1 ⇒ C(PC)
SNL	7420	1	Skip on non-zero L. If C(L) = 1, then C(PC) + 1 ⇒ C(PC)
SKP	7410	1	Skip, unconditional. The next instruction is skipped. C(PC) + 1 ⇒ C(PC)
OSR	7404	2	OR with switch register. (May be combined with CLA.) C(SR) V C(AC) ⇒ C(AC)
HLT	7402	2	Halt. Stops the program. If this instruction is combined with others in the OPR 2 group, the computer stops immediately after completion of the cycle in process.

THE ORGANIZATION OF MEMORY

A PDP-8 memory field can be likened to a book. The 4,096 words of the memory field correspond to the lines of text, and if we divide the memory into segments of 128 words each, we have an analogy to a 32-page book in which each page contains 128 lines.

The memory field is in fact segmented in this fashion, and the analogy to a book is affirmed by the fact that each of the 128-word blocks is called a page. Because the PDP-8 instruction word is not long enough to allow direct reference to all of the registers in a memory field, a special type of address reference must be provided. This can be illustrated by pursuing our book analogy a little bit farther.

Suppose that one is reading a text, and taking notes in the margins of each page. One can expect that some of the notes will refer to other parts of the page, or to information on other pages. For convenience, if one of the notes refers to a line of text on the same page, write only the line number. If the note refers to a line on some other page, write the page number followed by the number of the line on that page. Alternatively, of course, it is possible to begin on the first page and number all the lines of the book consecutively, and refer to them by these numbers alone.

In similar fashion, a given word of memory may be referred to by its page address; that is, its address within a page, or by its absolute address, which designates its position in the whole of memory. Here the analogy ends.

A PDP-8 memory field is organized as follows: the 4,096 words are arranged sequentially, with absolute addresses of 0 through 7777_8. The field is divided into 32 pages numbered from $0\text{-}37_8$. Each page contains 128 registers, with page addresses of $0\text{-}177_8$. As Figure 3-1 shows, the address field of a memory field of a memory reference instruction contains 7 bits, which is just enough to allow access to 200_8 locations. If bit 4 of the instruction contains a 1, the address field of the instruction refers to one of the addresses on the current page, that is, the page in which the instruction is stored. If bit 4 contains a 0, the reference is to an address on page 0.

The state of bit 3 of the instruction determines what is done with the contents of the memory register specified by bits 4-11. If this bit is 0, the contents of the cell addressed by the instruction are taken as the operand, and the operation is completed. In this case, the address specified by the instruction is the effective address of the instruction. If, however, bit 3 contains a 1, the contents of the cell addressed are treated, not as the operand, but as the 12-bit absolute address of the register containing the operand. In other words, the cell addressed contains the effective address of the instruction. In this way, a memory reference instruction can indirectly address any register in the memory field, regardless of which page it is on.

Thus, there are four ways, depending on the states of bits 3 and 4 of a memory reference instruction, in which the effective address may be obtained.

TABLE 3-4 EFFECTIVE ADDRESS CALCULATION

Bit 3	Bit 4	Effective Address
0	0	The operand is in page 0 at the address specified by bits 5 through 11.
0	1	The operand is in the current page at the address specified by bits 5 through 11.

TABLE 3-4 EFFECTIVE ADDRESS CALCULATION (continued)

Bit 3	Bit 4	Effective Address
1	0	The absolute address of the operand is taken from the contents of the location in page 0 designated by bits 5 through 11.
1	1	The absolute address of the operand is taken from the contents of the location in the current page designated by bits 5 through 11.

COMPLEMENT ARITHMETIC

In the PDP-8, as in other machines utilizing complementation techniques, negative numbers are represented as the complement of positive numbers, and subtraction is achieved by complement addition. Representation of negative values in 2's complement arithmetic is slightly different from that in 1's complement arithmetic.

The 1's complement of a number is the complement of the absolute positive value; that is, all 1's are replaced by 0's and all 0's are replaced by 1's. The 2's complement of a number is equal to the 1's complement of the positive value plus one.

In 1's complement arithmetic a carry from the sign bit (most significant bit) is added to the least significant bit in an end-around carry. In 2's complement arithmetic a carry from the sign bit complements the link (a carry would set the link to 1 if it were properly cleared before the operation), and there is no end-around carry.

A 1's complement representation of a negative number is always one less than the 2's complement representation of the same number. Differences between 1's and 2's complement representations are indicated in the following table.

TABLE 3-5 ONE'S AND TWO'S COMPLEMENT REPRESENTATIONS

Number	One's Complement	Two's Complement
+5	000000000101	000000000101
+4	000000000100	000000000100
+3	000000000011	000000000011
+2	000000000010	000000000010
+1	000000000001	000000000001
+0	000000000000	000000000000
-0	111111111111	Nonexistent
-1	111111111110	111111111111

TABLE 3-5 ONE'S AND TWO'S COMPLEMENT REPRESENTATIONS (continued)

Number	One's Complement	Two's Complement
-2	111111111101	111111111110
-3	111111111100	111111111101
-4	111111111011	111111111100
-5	111111111010	111111111011

Note that in 2's complement there is only one representation for the number which has the value zero, while in 1's complement there are two representations. Note also that complementation does not interfere with sign notation in either 1's or 2's complement arithmetic; bit 0 remains a 0 for positive numbers and a 1 for negative numbers.

PDP-8 arithmetic operations (as exemplified by the TAD instruction) are carried out in 2's complement. This means that the operands involved in the arithmetic must be in correct 2's complement representation. The 2's complement of any number is formed by taking the 1's complement, and adding 1 to it. The two operate class instructions, CMA and IAC, may be combined into a single instruction to perform the operation of taking a 2's complement. Because the operation is often required, the mnemonic CIA is given to the combined instruction.

Addition

The addition of a number contained in a core memory location and the number contained in the accumulator is performed directly by using the TAD Y instruction, assuming that the binary point is in the same position and that both numbers are properly represented in 2's complement arithmetic. Addition can be performed without regard for the sign of either the augend or the addend. Overflow is possible, in which case the result will have an incorrect sign, although the eleven least significant bits will be correct. A carry from bit 0 will complement the link bit.

Subtraction

Subtraction is performed by complementing the subtrahend and adding the minuend. As in addition, if both numbers are represented by their 2's complement, subtraction can be performed without regard for the signs of either number.

CHAPTER 4

ORGANIZATION

The rules for writing a program can best be introduced by tracing the steps from the statement of a problem to the completed routine. Consider, for example, a program to compute the sum of the first n integers,

$$s = n + (n-1) + (n-2) + (n-3) + \ldots + 3 + 2 + 1$$

First, a flow chart, that is, a block diagram of the general steps to the solution of the problem, should be made. It might look something like Figure 4-1. Ignoring the first box for the moment, consider the second, third, and fourth boxes. The operations specified in these three boxes perform the main computation of the sum. Box 2 specifies the actual arithmetic of computing the partial sum. Box 3 is a counter, which counts the number of partial additions that have been made. Box 4 is a decision point; if the count indicates that the sum is complete, the program goes on to box 5; if it is not complete, the program returns to the beginning of the main computation. This sort of continuous recycling through a section of program is called a loop.

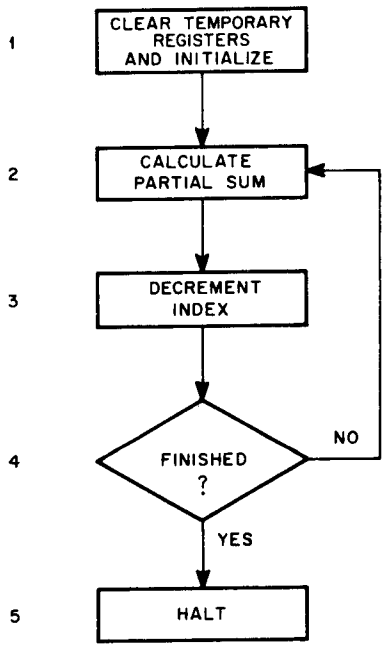

Figure 4-1 Flow Chart of Program to Calculate Sum of Integers

The first box in this chart specifies the operations which prepare the program for operation. Since nearly every program written will be used more than once, with different data each time, it is necessary at the

start of the routine to clear out old results from previous use of the program so that one can start fresh with new data, in the same way that one clears the keyboard and registers of an adding machine before starting a new calculation. In programming terms, this preparation is called initializing.

Besides indicating the general sequence of operations, the flow chart also gives an idea of storage requirements. Space will be needed not only for the instructions which perform the computation, but also for the data used by the routine. One register will be needed to hold the partial sum accumulated through repeated additions; this register will naturally hold the correct total when the calculation is finished. Another cell is needed for the index, which is the counter specified in box 3 of the flow chart. Finally, one register is necessary to hold the number n, which determines the limit of the computation.

The program, then, must supply two types of information. First, it must include the executable instructions, data, and temporary registers which will occupy memory cells when the program is executed. Second, it must include a certain amount of information for the MACRO Assembler itself, to establish the locations and extent of the program in memory. These assembler directives are called pseudo-instructions; they are included in the program along with the executable instructions. A pseudo-instruction is rather like a proofreader's mark on a manuscript; the mark provides information to the editor, but does not itself cause additional text to appear in the printed book. Similarly, a pseudo-instruction provides information to the Assembler, but does not itself cause any coding to be inserted into the object program.

CODING

Now to write the program. First one must decide where the program is to be stored. To do this, it is necessary only to establish the location of the first register of the program. This word is called the origin, and can be designated in two ways. The most common way is to designate the page in which the program is to be stored; the origin is automatically set at the first register (page address 0) in that page. Thus, to put the routine in page 2, the pseudo-instruction, PAGE, would be used as follows:

```
PAGE 2
```

The program would then begin in location 0 of page 2.

To set the origin to any register other than the first one in a page, the absolute address of the starting location must be specified. This is done by using the special character "*". To place the origin at location 210, the coding would be:

```
*210
```

In establishing the origin, an index within the Assembler, called the current location counter (CLC), has been set equal to the absolute address of the origin. For example, the use of PAGE 2 would set the CLC

to a value of 400, which is the absolute address of the first register in page 2. Likewise, *210 causes the CLC to be set to a value of 210. In programming terms the CLC points to the origin; such an index is called a pointer.

The coding for the program may now be written. First three registers for the three items of data must be reserved. Usually, data storage is set aside by placing a 0 in each register, thus allocating a cell. The coding looks like this:

```
PAGE 2
      0
      0
      0
```

Since the origin has been set at location 400, the three data cells occupy locations 400, 401, and 402 (page addresses 0, 1, and 2, respectively).

Next, write the coding for the computation.

```
PAGE 2
0
0
0
CLA
DCA 400
TAD 401
DCA 402
TAD 400
TAD 402
DCA 400
STA
TAD 402
SZA
JMP 406
HLT
$
```

The program now occupies locations 400-416 (0-16 on page 2).

The special character $ indicates to the Assembler that it is a complete program and that nothing else is to follow.

This program is now complete, and can be assembled into a working routine. The coding, however, is rather stark, and not very useful to another programmer, should he wish to discover what the program does. Of course, he could figure out the function of the routine by going through it step by step with pencil

and paper, but if this program were much longer, this task would be tedious and impractical. Comments, a method of explaining the functions of the program, make the work immediately useful to someone else.

COMMENTS

Comments are included in a PDP-8 program in a simple way. Consider the three data storage locations of our program. A comment would identify each register's function immediately; they might appear as:

```
...
0                    /PARTIAL SUM AND FINAL TOTAL
0                    /INTEGER N
0                    /INDEX
...
```

The slash preceding each comment is the character which identifies a comment field. All information on that line following the slash is considered to be a comment, and is ignored by the Assembler. For clarity and neatness, a tab has been inserted between the instruction and the comment.

Since a slash identifies all the subsequent information on a line as a comment, one full line can be used for a title by placing the slash in the first space after the left margin. With a title, the program might look like this:

```
/INTEGER SUMMATION ROUTINE
PAGE 2
      0              /PARTIAL SUM AND FINAL TOTAL
      ...
      ...
```

Now, an immediate identification of what the program is to be used for is included in the listing.

More comments explain the workings of the program:

```
/INTEGER SUMMATION ROUTINE
PAGE 2
      0              /PARTIAL SUM AND FINAL TOTAL
      0              /INTEGER N
      0              /INDEX
CLA
DCA 400              /ZERO TO SUM
TAD 401
DCA 402              /SET INDEX
TAD 400              /MAIN LOOP
TAD 402
DCA 400
STA                  /DECREMENT INDEX
```

```
TAD 402
SZA                     /IS IT 0?
JMP 406                 /NO:  KEEP GOING
HLT                     /YES:  HALT
$
```

Now it is a little easier to understand the program, but some limitations remain. It is tied to one particular place in the computer's memory; to put it in some other location, recoding the entire routine with new addresses would be necessary. Nor are these octal page addresses much help in keeping track of program flow; even in a short routine such as this, it is necessary to do some counting to find out what location the instruction STA is in, for example. A simple, meaningful way of identifying storage addresses is needed.

ADDRESS TAGS

As described above, a symbol is assigned to each of the PDP-8 instruction codes. Similarly, the programmer can assign a symbol to any one of the storage addresses in the computer memory. For instance, the routine under discussion contains three important registers: the beginning of data storage, the beginning of the program sequence, and the beginning of the main loop.

Each one of these locations can be labelled with a symbolic tag, in the following way:

```
/INTEGER SUMMATION ROUTINE
PAGE 2
DATA, 0                 /PARTIAL SUM AND FINAL TOTAL
      0                 /INTEGER N
      0                 /INDEX
BEGIN, CLA
       DCA 400          /ZERO TO SUM
       TAD 401
GO,    DCA 402          /SET INDEX
       TAD 400          /MAIN LOOP
       TAD 402
       DCA 400
       STA              /DECREMENT INDEX
       TAD 402
       SZA              /IS IT 0?
       JMP 406          /NO:  KEEP GOING
       HLT              /YES; HALT
       $
```

Three reference points have been established. Note that the tags are roughly mnemonic, thus offering an additional measure of program identification and clarification. In programming terms, three symbols have been defined. In this case, each symbol has a value that is equal to the absolute address of the register in which the associated item (data word or instruction) is stored. Thus, the value of the symbol DATA is 400; of BEGIN, 403; of GO, 406.

Symbolic Addressing

Locating sections of a program has been simplified, but full advantage of address tags has not yet been taken. The instruction addresses themselves are still tied to the absolute addresses of the program. However, since each tag has a numeric value corresponding to an address, a tag may also be used as a symbol in the address part of an instruction. For example, the absolute address 402 is clearly equivalent to the expression, BEGIN-1 (BEGIN=403; 403-1=402). Similarly, the address 411 is equivalent to GO+3. Replacing absolute addresses with symbolic expressions, the program will look like this:

```
/INTEGER SUMMATION ROUTINE
PAGE 2
DATA, 0                     /PARTIAL SUM AND FINAL TOTAL
      0                     /INTEGER N
      0                     /INDEX
BEGIN, CLA
       DCA DATA             /ZERO TO SUM
       TAD DATA+1
GO,    DCA DATA+2           /SET INDEX
       TAD DATA             /MAIN LOOP
       TAD DATA+2
       DCA DATA
       STA                  /DECREMENT INDEX
       TAD DATA+2
       SZA                  /IS IT 0?
       JMP GO               /NO: KEEP GOING
       HLT                  /YES: HALT
       $
```

This program that is easily readable, includes a running commentary for explaining its functions, and carries its own symbolic references that further assist in understanding and keeping track of the routine. Moreover, the routine has been freed from a specific place in memory by the use of symbolic addresses. To change the location of the program, it is necessary only to change the origin setting by the PAGE pseudo-instructions.

STORAGE TECHNIQUES

The program above will function but it may be improved. One of its drawbacks is the fact that wherever the program is stored, the data must be stored right along with it. When space is at a premium, as it often is, it would be desirable to put all of the data in a fixed place, and let the working parts of the program be arranged as necessary.

In the PDP-8, data that is used by several parts of a program is often stored in page 0. If each of the three data words in the program is given a name, page 0 addresses can be assigned to them. Call the first

word TOTAL, the second INDEX, and the third N. In effect, the data words have been labeled with address tags, but since the names can be used in symbolic expressions (for instance, in an instruction address), programmers usually refer to them as variables. If the variables are assigned to page 0 and the instruction address in the program changed to match, the coding will look like this:

```
/INTEGER SUMMATION ROUTINE
*20
TOTAL, 0                        /THESE REGISTERS ARE
INDEX, 0                        /NOW ON PAGE 0
N,     0
PAGE 2
BEGIN, CLA
       DCA TOTAL                /ZERO TO SUM
       TAD N
GO,    DCA INDEX                /SET INDEX
       TAD TOTAL                /MAIN LOOP
       TAD INDEX
       DCA TOTAL
       STA                      /DECREMENT INDEX
       TAD INDEX
       SZA                      /IS IT 0?
       JMP GO                   /NO: KEEP GOING
       HLT                      /YES: HALT
       $
```

When the program is loaded, the three items of data (having been assigned addresses on page 0) will be stored in page 0.

A large step toward our goal of a refined, useful routine has been taken; i.e., a program which may be placed anywhere in memory and is easily understood and interpreted by someone other than the author, because of liberal comments and symbolic addressing.

OPTIMUM USE

It remains only to relate this sequence of coding to the context in which it is likely to operate. Obviously the summation program is useless in core by itself, waiting for an integer summation to be carried out.

Since the example has served its purpose in illustrating how a complete program can be written, all the coding except the minimum necessary for the actual computation will be removed, leaving this (the ellipsis always indicates the presence of unspecified coding):

```
.
.
.
BEGIN, CLA
       DCA TOTAL          /ZERO TO SUM
       TAD N
GO,    DCA INDEX          /SET INDEX
       TAD TOTAL          /MAIN LOOP
       TAD INDEX
       DCA TOTAL
       STA                /DECREMENT INDEX
       TAD INDEX
       SZA                /IS IT 0?
       JMP GO             /NO: KEEP GOING
       HLT                /YES: HALT
```

This sequence can now be placed within a larger program wherever it may be needed. Figure 4-2 is an example of a long program which uses this routine two times. Whenever the summation routine occurs, it is preceded by an instruction which takes the value of N from the AC and places it on the proper location in data storage. One other change is also necessary. The address tag GO has been eliminated from all but the first occurrence in the coding sequence; it is obvious that one symbol cannot be defined with more than one value, or there would be confusion as to which of the values was meant at any one time. As a result, it is necessary to eliminate the reference to GO in the JMP instruction at the end of the sequence.

Notice that the instruction, DCA INDEX always remains exactly seven locations ahead of the JMP instruction, regardless of where the routine is stored. Evidently, an address expression which could refer to the location of DCA INDEX relative to the location of the JMP instruction is needed. In other words, a construction is desired that will perform the same function as the following but without having to define the symbol HERE:

```
       DCA INDEX
       TAD INDEX
HERE,  JMP HERE-2
```

Remembering that the current location counter always "points" to the location of the instruction currently being assembled, it becomes apparent that what is desired is a symbol which, whenever it is used, always takes the value of the CLC. In MACRO-8, this symbol is the period, ".". Whenever this character is encountered in an address expression, the value of the CLC is substituted for it. In the example in Figure 4-2 it is used in the address of the JMP instruction wherever it appears in the routine. In this way, the necessity of having to define a new set of symbols each time the routine is used is avoided.

```
PAGE 10

                ...
                ...
BEGIN,    DCA     N                   /INITIALIZATION
          DCA     TOTAL
          TAD     N
GO,       DCA     INDEX               /SET INDEX
          TAD     TOTAL               /MAIN LOOP
          TAD     INDEX
          DCA     TOTAL
          STA                         /INCREMENT INDEX
          TAD     INDEX
          SZA                         /IS IT 0?
          JMP     GO                  /NO: KEEP GOING
          ...                         /YES: ALL DONE
          ...

          DCA     N                   /INITIALIZATION
          DCA     TOTAL
          TAD     N
          DCA     INDEX               /SET INDEX
          TAD     TOTAL               /MAIN LOOP
          TAD     INDEX
          DCA     TOTAL
          STA                         /DECREMENT INDEX
          TAD     INDEX
          SZA                         /IS IT 0?
          JMP     .-7                 /NO: KEEP GOING
                                      /YES: ALL DONE
          ...
          ...
          ...
$
```

Figure 4-2 Program Example

SUBROUTINES

Now the program example is useful, but one more difficulty must be overcome. An examination of Figure 2-4 will make this problem obvious: If the routine is used very many times, an extremely large amount of storage space will be used simply in repetitions ot the same coding sequence. This counteracts the general aim of writing a concise, efficient program. If the routine could be written only once, placed in a separate section of memory and called into operation each time it was needed, the whole procedure would be more efficient.

A program that may be used in this manner is called a subroutine. It is a sequence of coding with the following properties:

1. It is self-contained, that is, it can be assembled by itself without having to be part of a larger program.

2. It occupies its own section of memory, logically separate from other coding sequences.

3. It performs only one task. Each subroutine has a definite purpose.

4. It is called into operation only by another program, and when it has finished its task, it returns to that program. It can be called any number of times, and upon completion always returns control to the point from which it was last called.

5. When necessary, it uses data supplied by the calling program, and returns results to a place accessible by that program.

When a subroutine is written, four requirements which are implied by the last two properties listed above must be fullfilled:

a. The data must be accessible by the subroutine.
b. The results must be accessible by the calling program.
c. There must be a way of calling the subroutine into operation.
d. There must be a way of returning control to the calling program.

The problem of data transmission has been solved by placing the three data storage registers in page 0, thereby making them accessible to any program or subroutine in memory. The transfer and control of return is not quite so obvious. First, restore the symbolic tags to the subroutine as well as the title and other pseudo-instructions to make it a self-contained program giving:

```
/INTEGER SUMMATION SUBROUTINE
PAGE 2
INTSUM, DCA N                   /SAVE INPUT IN C(AC)
        DCA TOTAL               /ZERO TO SUM
        TAD N
GO,     DCA INDEX               /SET INDEX
        TAD TOTAL               /MAIN LOOP
        TAD INDEX
        DCA TOTAL
        STA                     /DECREMENT INDEX
```

```
        TAD INDEX
        SZA                     /IS IT 0?
        JMP GO                  /NO: CONTINUE
        ....                    /YES
$
```

At a first glance, it could seem easy to eliminate the problem of transferring control to the subroutine. A JMP INTSUM in the calling program every time the subroutine was called would provide the path. But what happens at the end of the calculation? The subroutine has no way of knowing where to return. The subroutine cannot merely halt as this violates requirement d above.

The solution lies in the use of the JMS instruction. Remembering from Chapter 2 when the JMS instruction is executed, the contents of the program counter are saved in the location addressed by the instruction, and control is transferred to the register immediately following the one addressed. Thus, a register which contains the address of the next instruction following the JMS is available. This can be used to return to the proper point in the calling program when the subroutine has finished its computation. Rearrange the first part of our subroutine as follows:

```
/INTEGER SUMMATION ROUTINE
PAGE 2
INTSUM, 0                       /SAVE PC HERE
        DCA N                   /SAVE INPUT NUMBER
        DCA TOTAL               /ZERO TO SUM
        TAD N
GO,     DCA INDEX               /SET INDEX
        ...
        ...
```

Now, whenever the instruction JMS INTSUM is used, the contents of the program counter are stored in the location tagged with the name INTSUM and the computer continues operation with the instruction immediately following which is the first instruction of the subroutine.

Now that the subroutine has been called, a means to exit from it must be provided. The JMS instruction stores the program counter which "points" to the instruction following the JMS. After the execution of a JMS INTSUM, the register INTSUM contains the contents of the program counter at the time the JMS was executed. This can be used as an effective address (See Chapter 3, the "Organization of Memory") to return to the point of call. To do this, the subroutine may be terminated with a JMP instruction which references the register INTSUM as an indirect address as follows:

```
/INTEGER SUMMATION SUBROUTINE

PAGE 2
INTSUM, 0                       /SAVE PC HERE
        DCA N                   /SAVE INPUT NUMBER
        ...
        ...
        JMP I INTSUM            /YES:  EXIT SUBROUTINE
```

The character "I" is used to signify indirect addressing. In use here it means, "Jump to the register whose address is contained in the register tagged INTSUM."

The complete subroutine is shown below: (not including the page 0 definition of the variables).

```
/INTEGER SUMMATION SUBROUTINE

PAGE 2
INTSUM, 0                       /SAVE PC HERE
        DCA N                   /SAVE INPUT NUMBER
        DCA TOTAL               /ZERO TO SUM
        TAD N
GO,     DCA INDEX               /SET INDEX
        TAD TOTAL               /MAIN LOOP
        TAD INDEX
        DCA TOTAL
        STA                     /DECREMENT INDEX
        TAD INDEX
        SZA                     /IS IT 0?
        JMP GO                  /NO:  CONTINUE
        JMP I INTSUM            /YES:  ALL DONE
```

The reader is referred to the PDP-8 Library for other examples of subroutine calling techniques.

The MACRO-8 Programming Language may now be discussed in detail.

PART 2

THE MACRO LANGUAGE

CHAPTER 5

MACRO-8 PROGRAMMING LANGUAGE

The MACRO-8 Symbolic Assembler accepts source programs written in symbolic language and translates them into binary form. MACRO-8 produces an object program tape (binary), a symbol table defining memory locations (for use with DDT), an octal/symbolic assembly listing, and useful diagnostic messages. MACRO-8 is compatible with PAL III, but has the following additional features:

User-Defined Macro's	Groups of computer instructions required for the the solution of a specific problem can be defined as a macro instruction by the user.
Double Precision Integers	Positive or negative double precision integers are allotted two consecutive core locations.
Floating Point Constants	The format and rules for defining these constants are compatible with the format used by the PDP-8 Floating Point Package (See Digital-8-5-S).
Operators	Symbols and integers may be combined with a number of operators. + Addition & Boolean AND - Subtraction ! Boolean Inclusive OR
Literals	Symbolic or integer literals (constants) are automatically assigned.
Text Facility	There are text facilities for single characters and blocks of text.
Link Generation	Links are automatically generated for out of page references.

To incorporate these new features, it was necessary to decrease the size of the symbol table and because of this, programs that were originally coded to be assembled by PAL III might have too many symbols to be assembled by MACRO-8.

CHARACTERS

Programs in the MACRO-8 language are written using characters from the ASCII character set. The following characters are used:

Letters

A B C D . . . X Y Z

Digits

1 2 3 4 5 6 7 8 9 0

Punctuation Characters

Since a number of characters are invisible (i.e., nonprinting), the following notation is used to represent them in the examples:

␣	space
→|	tab
↵	carriage return

The following characters are used to specify operations to be performed upon symbols or numbers:

Character		Use
␣	space	Combine symbols or numbers
+	plus	Combine symbols or numbers (add)
−	minus	Combine symbols or numbers (subtract)
!	exclamation point	Combine symbols or numbers (OR)
↵	carriage return	Terminate line
→|	tab	Combine symbols or numbers or format source tape
,	comma	Assign symbolic address
=	equals	Define parameters
;	semicolon	Terminate coding line
$	dollar sign	End of pass
*	asterisk	Set location counter
.	point	Has value equal to current location counter

/	slash	Indicates start of a comment
&	ampersand	Combine symbols or numbers (AND)
"	quote	Generate ASCII constant
( )	parentheses	Define literal on current page
[ ]	brackets	Define page 0 literal
< >	angle brackets	Define a macro

Ignored Characters

Form-feed	End of a logical page of a source program (see Symbolic Editor)
Blank tape	Used for leader/trailer
Rubouts	Used for deleting characters
Code 200	Used for leader/trailer
Line-feed	Follows carriage-return

All other characters are illegal when not in a comment or a TEXT field, and cause the error message IC to be printed. The form-feed is used at the end of a page of program for editing purposes. The functions of leader and trailer are self-explanatory. This may be either blank tape or code 200.

Tabulations are usually used in the body of a program to provide a neat page for printing; they can separate fields from one another, as between an instruction and an associated comment. For example, a line could be written as:

```
GO,     TAD TOTAL/MAIN LOOP)
```

but it is far easier to read if tabs are inserted:

```
GO, →| TAD TOTAL →| /MAIN LOOP)
```

(the characters →| and) are nonprinting)

Either ; (semicolon) or the combination carriage-return/line-feed may be used as line terminators. The semicolon is considered identical to carriage-return/line-feed except that it will not terminate a comment. Example:

```
TAD A                          /THIS IS A COMMENT  ;  TAD B)
```

The entire expression up to the carriage return is considered a comment.

As was noted previously, the tabulation is used as a formatting device to provide a neat appearance for the printed program listing. Use of the semicolon allows the programmer to place several lines of coding on a single line. If, for example, he wishes to write a sequence of instructions to rotate the C(AC) and C(L) six places to the right, it might look like:

```
...
RTR)
RTR)
RTR)
...
```

He may place all three instructions as a single line by substituting the control character ";" (semicolon) for the line terminator ") " (carriage return). The above sequence of instructions may be rewritten as:

```
RTR;  RTR;  RTR)
```

This type of format is particularly useful when setting aside a section of data storage for a list. For example, a 12-word list could be reserved by:

```
LIST,   0;      0;      0;      0;      0;      0)
        0;      0;      0;      0;      0;      0)
```

A neat printout (or program listing, as it is usually called) makes subsequent editing, debugging, and interpretation much easier than if the coding were laid out in a haphazard fashion. The coding practices listed below are in general use, and will result in a readable, orderly listing. (See Digital-8-21-U for a program to produce listings of this form.)

1. A title comment begins at the left hand margin.

2. Pseudo-instructions may begin at the left margin; often, however, they are indented one tab stop to line up with the executable instructions.

3. Address tags begin at the left margin. They are separated from succeeding fields by a tabulation.

4. Instruction fields, whether or not they are preceded by a tag field, are indented one tab stop.

5. A comment is separated from the preceding field by a tabulation, unless it occupies the whole line, in which case it usually begins at the left margin.

Elements

Any group of letters, digits, and punctuation characters which represents binary values less than 2^{12} is an element.

Integers

Any sequence of numbers delimited by punctuation characters forms a number. Example:

1
12
4372

The radix control pseudo-instructions indicate to the Assembler the radix to be used in number interpretation. The pseudo-instruction DECIMAL indicates that all numbers are to be interpreted as decimal until the next occurrence of the pseudo-instruction OCTAL.

The pseudo-instruction OCTAL indicates that all numbers are to be interpreted as octal until the next occurrence of the pseudo-instruction DECIMAL. The radix is initially set to octal and remains octal unless otherwise specified.

Symbols

A symbol is a string of one or more alphanumeric characters delimited by a punctuation character. Symbols are composed according to the following rules:

1. The characters must be either alphabetic (A-Z) or numeric (0-9).

2. The first character must be alphabetic.

3. Only the first six characters of any symbol are meaningful; the remainder, if any, are ignored.

4. A symbol is terminated by any nonalphanumeric character.

The MACRO Assembler has, in its permanent symbol table, definitions of the symbols for all PDP-8 operation codes, operate commands, and many IOT commands (see Appendix A for a complete list). These may be used without prior definition by the user. Example:

JMS — is a symbol whose value of 4000 is taken from the operation code definitions.

A — is a user-created symbol. When used as a symbolic address tag, its value is the address of the instruction it tags. This value is assigned by the Assembler.

Note that because of rule 3, a symbol such as INTEGER, for instance, would be interpreted as INTEGE since the seventh letter is ignored. If symbols of more than six characters are used, the programmer should be careful to avoid the error of defining two apparently different symbols whose first six characters are, in fact, identical. For example, the two symbols, GEORGE1 and GEORGE2, differ only in the seventh character, so that the Assembler would treat them as being the same symbol, GEORGE.

It is not necessary to define a symbol before it is used in an expression. They must be defined before the end of PASS 1, however. Thus, one may refer to a number of registers by their address tags, and then define the symbols later.

Parameter Assignments

A symbol may be assigned a value by means of a parameter assignment statement which looks like an algebraic statement. The single symbol to the left of the equal sign is assigned the value of the expression on the right. No space(s) or tab(s) may appear between the single symbol to the left of the equal sign and the equal sign. Examples:

```
A = 6
EXIT = JMP I 0
C = A + B
```

All symbols to the right of an "=" sign must be already defined. The symbol to the left of the "=" sign and its associated value is stored in the Assembler's symbol table.

The use of the "=" does not increment the current location counter. It is, rather, an instruction to the Assembler itself rather than a part of the output binary. The equal sign may be used to redefine a symbol.

Symbol Definition

A symbol may be defined by the user in one of three ways:

1. By use of a parameter assignment. Example:

```
DISMIS = JMP RESTOR
```

2. As a macro name. Example:

```
DEFINE LOAD A
< CLA
  TAD A >
```

3. By use of the comma. When a symbol is terminated by a comma, it is assigned a value equal to the current location counter. Example:

```
      *100                    /SET CLC TO 100
TAG,  CLA
      JMP A
B,    0
A,    DCA B
      ...
      ...
```

The symbol "TAG" is assigned a value of 0100, the symbol "B" a value of 0102, and the symbol "A" a value of 0103.

Expressions

All elements, i.e., symbols and numbers (exclusive of pseudo-instruction symbols, macro names, and double precision or floating point constants) may be combined with certain operators to form expressions. These operators are:

+	plus	This signifies 2's complement addition (modulo 4096_{10}).
–	minus	This signifies 2's complement subtraction (modulo 4096_{10}).
!	exclamation point	This signifies Boolean inclusive OR (union).
&	ampersand	This signifies Boolean AND (intersection).
␣	space	Space is interpreted in context. It may signify inclusive OR or act as a field delimiter.

Symbols and integers may be combined with any of the above operators. A symbolic expression is evaluated from left to right; no grouping of terms is permitted. Example:

	A	B	A+B	A–B	A ! B	A&B
Value	0002	0003	0005	7777	0003	0002
Value	0007	0005	0014	0002	0007	0005
Value	0700	0007	0707	0671	0707	0000

The MACRO-8 Assembler makes a distinction between the types of symbols it is processing. These types are 1) permanent symbols, 2) user defined symbols, and 3) macro names. The character "space" is interpreted written in the context of the expression. If a space is used to delimit two or more permanent symbols, space signifies inclusive OR. Example:

```
CLA     is a permanent symbol whose value is 7200.
CMA     is a permanent symbol whose value is 7040.
```

The expression:

```
CLA ␣ CMA     has a value of 7240.
```

If the symbol following the space is a user defined symbol, space acts as an address field delimiter. Example:

```
    *2117
A,      CLA
        ...
        ...
        JMP ␣ A
```

"A" is a user defined symbol whose value is 2117. The expression JMP ␣ A is evaluated as follows:

The seven address bits of A are taken, i.e.:

```
A 010 00 1 001 111
          1 001 111
```

The remaining five bits of A are tested to see if they are 0's (page 0 reference); if they are not, the current page bit is set.

```
000 011 001 111
```

The operation code is ORed into the expression:

```
     JMP  101 000 000 000
Address A 000 011 001 111
JMP    A  101 011 001 111
```

or, written more concisely:

```
5317
```

In addition to the above outlined tests, the page bits of the address field are compared with the page bits of the current location counter. If the page bits of the address field are nonzero and do not equal the page bits of the current location counter, an out-of-page reference is being attempted. If the reference is to an address not on the page where the instruction will be located, the Assembler will set the indirect bit (bit 3) and an indirect address linkage will be generated on the current memory page. If the out-of-page reference is already an indirect one, the error diagnostic II (Illegal Indirect) will be typed on PASS 2. When the link is generated, the LG (Link Generated) message will be typed on PASS 2. In the case of several out-of-page references to the same address, the link will be generated only once, but the LG message will be printed each time. Example:

```
*2117
A,    CLA
        .
        .
        .
*2600
      JMP ␣ A
```

The space preceding the user defined symbol "A" acts as an address field delimiter. The Assembler will recognize that the register tagged "A" is not on the current page (in this case 2600-2777) and will generate a link to it as follows:

in location 2600 the Assembler will place the word
5777 which is JMP I 2777
in address 2777 (the last location on the current page), the word 2117 (the actual address of "A") will be placed.

The address field of a memory reference instruction may be any valid expression. Example:

```
A=270
*200
TAD A-20
```

would produce, in location 200, the word

001 010 101 000 or 1250 (TAD 250)

Although the Assembler will recognize and generate an indirect address linkage when necessary, the programmer may indicate an explicit indirect address by using the special symbol "I". This must be between the operation code and the address field. The Assembler cannot generate a link for an instruction that is already specified as being an indirect reference. In this case, the Assembler will type the message II (Illegal Indirect).

Current Address Indicator

The single character period (.) has, at all times, a value equal to the value of the current location counter. It may be used as any integer or symbol (except to the left of an equal sign). Example:

```
*200
JMP .+2
```

Is equivalent to JMP 202.

```
*300
.+2400
```

would produce, in register 0300, the quantity 2700.

```
    *2200
CALL=JMS I .
    0027
```

Since the second line, CALL=JMP I ., does not increment the current location counter, 0027 would be placed in register 2200 and CALL would be placed in the symbol table with an associated value of

100 110 000 000 or 4600.

Origin Setting

The origin (current location counter) is reset by use of the special character asterisk (*). The current location counter is set to the value of the expression following the "*". The origin is initially set to 0200. All symbols to the right of "*" must already have been defined. Example:

If D has the value 250

then

*D+10 will set the location counter to 0260.

To simplify page handling, the pseudo-instruction PAGE may be used. When "PAGE" is encountered, the origin is reset to the first location of the next page. A page number may be specified by a legal expression following the page pseudo-instruction. Example:

```
*270
...
...
```

at this point, either

```
*400
```

or

```
PAGE
```

or

```
PAGE 2
```

will reset the origin to 0400.

Literals

Since the symbolic expressions which appear in the address part of an instruction usually refer to the locations of registers containing the quantities being operated upon, the programmer must explicitly reserve the registers holding his constants. The MACRO-8 language provides a means for using a constant directly. Suppose, for example, that the programmer has an index which is incremented by two. One way of coding this operation would be as follows:

```
        ...
        CLA
        TAD INDEX
        TAD C2
        DCA INDEX
        ...
        ...
C2, 2
```

Using a literal, this would become

```
        ...
        CLA
        TAD  INDEX
        TAD  (2)
        DCA  INDEX
        ...
```

The left parenthesis is a signal to the Assembler that the expression following is to be evaluated and assigned a register in the constants table of the current page. This is the same table in which the indirect address linkages are stored. In the above example, the quantity 2 is stored in a register in a list beginning at the top of the memory page (page address 177), and the instruction in which it appears is encoded with an address referring to that address. A literal is assigned to storage the first time it is encountered; subsequent references will be to the same register.

If the programmer wishes to assign literals to page 0 rather than the current page, he may use square brackets, "[" and "]", in place of the parentheses. However, in both cases, the right of closing member may be omitted. The following examples are acceptable:

```
TAD   (777
AND   [JMP
```

Note that in the second example, the instruction AND [JMP has the same effect as AND [5000.

Literals may be nested. For example:

```
*200
TAD   (TAD   (30
```

will generate

```
0200          1376
...           ...
0376          1377
0377          0030
```

This type of nesting may be carried to as many levels as desired. Literals are stored on each page starting at page address 177 and extending toward page address 0. (Only 127_{10} or 177_8 literals may be placed on page 0). If a literal is generated for a nonzero page and then the origin is set to another page, the current page literal buffer is punched out (during PASS 2). If the origin is then reset to the previously used page, the same literal will be generated if used again.

Single Character Text Facility

If a single character is preceded by a double quote, the 8-bit value of the ASCII code for the character is inserted instead of taking the letter as a symbol. Example:

```
CLA
TAD   ("A
...
```

will place the constant 0301 in the accumulator.

CHAPTER 6

PSEUDO-INSTRUCTIONS

The pseudo-instructions are directions to the Assembler to perform certain tasks or to interpret subsequent coding in a certain way. By themselves pseudo-instructions do not generate coding or (in general) effect the current location counter. The functions of each pseudo-instruction are described in this chapter.

CURRENT LOCATION COUNTER

PAGE — This pseudo-instruction is used to set the current location counter.

PAGE n — This will reset the current location counter to the first address of page n, where n is an integer, a previously defined symbol, or a symbolic expression. Examples:

PAGE 2 will set the CLC to 0400
PAGE 6 will set the CLC to 1400

PAGE — When used without an argument, PAGE will reset the CLC to the first location on the next succeeding page. Thus, if a program is being assembled into page 1 and the programmer wishes to begin the next segment on page 2, he need only insert the pseudo-instruction PAGE, as follows:

```
        JMP     .-7
PAGE
        CLA
```

The current location counter may be explicitly set by use of the asterisk.

EXTENDED MEMORY

On PDP-8's equipped with more than one memory bank, the pseudo-instruction

FIELD n — may be used where n is an integer, a previously defined symbol, or a symbolic expression within the range $0 \leq n \leq 7$.

This pseudo-instruction causes a word of the form

11 XXX 000 where $000 \leq XXX \leq 111$

to be punched on the binary tape during PASS 2. This word is interpreted by the Extended Memory Binary Loaders (see Digital-8-2A-U; Digital-8-2B-U).

RADIX CONTROL

Normally, all integers used in a program are taken as octal numbers. If, however, the programmer wishes to have certain numbers treated as decimal, he may use the pseudo-instructions:

DECIMAL — When this pseudo-instruction occurs, all integers encountered in subsequent coding will be taken as decimal until the occurrence of the pseudo-instruction

OCTAL — which will reset the radix to its original (octal) base.

NUMBERS

The types of numbers allowed are integers (See Chapter 5), double precision integers, and double precision floating point numbers.

Double Precision Integers

Double precision integers may be positive or negative (2's complement) according to their sign but may not be combined with operators. They are always taken as decimal radix although the current radix is not disturbed. Each double precision integer is allotted two consecutive registers with the sign indicated by bit 0 of the first word.

The double precision integer mode is entered through the use of pseudo-instruction DUBL and all numbers encountered will be taken as double precision integers until an alphabetic character is encountered. Each number is terminated by the carriage return (↵) or the semicolon (;) or by a comment. Example:

```
          *400↵
DUBL      679467↵
          44↵
          -3↵
TAG, CLA↵
          ...
```

would produce

```
0400      0245
0401      7053
0402      0000
0403      0054
```

```
0404        7777
0405        7775
0406        7200
```

and the symbol "TAG" would have a value equal to 0406.

Floating Point Constants

Double precision floating point constants may be positive or negative according to their sign but may not be combined with operators. Decimal radix is assumed but the current radix is not altered. Floating point constants are each assigned three registers and are stored in normalized form. (See Digital-8-5-S for a description of floating point arithmetic.)

The double precision floating point mode is entered through use of the pseudo-instruction FLTG. All numbers encountered after the use of FLTG will be taken as double precision floating point constants until the occurrence of an alphabetic character other than E. The general input format of a floating point number is

±ddd.ddddE±dd

where the d's are decimal digits. Any character which is not legally part of the above format (except rubouts) terminates input of the number. Example:

		Produces	
	*400↵		
FLTG	+509.32E-02↵	0400	0003
		0401	2427
		0402	6670
	-62.97E04↵	0403	0024
		0404	5462
		0405	0740
	1.00E-2↵	0406	7772
		0407	2436
		0410	5574
TAG2,	CLA↵	0411	7200

and the symbol "TAG2" would be assigned a value of 0411.

TEXT FACILITY

There is a text facility for single characters and text strings. For a description of the single character mode (double quote), see Chapter 5.

A string of text may be entered by giving the pseudo-instruction TEXT followed by a space, a delimiting character, a string of text, and repeating the same delimiting character. Example:

```
TEXT    ATEXTA
```

The character codes are stored two to a register in ASCII code that has been trimmed to six bits. Following the last character, a 6-bit zero is inserted as a stop code. The above statement would produce

```
2405
3024
0000
```

```
TEXT ␣   /BOB/
```

would produce

```
0217
0200
```

The TEXT pseudo-op could also be used as part of a calling sequence to a subroutine:

```
a.          JMS MESS
            TEXT /          /
```

or

```
b.          JMS MESS
            NOWDS           /NO WDS IN MESSAGE
            ADDMESS         /ADDRESS OF MESSAGE
                .
                .
                .
                .
ADDMESS,    TEXT /          /
```

Note that while the TEXT pseudo-instruction causes characters to be stored in a trimmed code, the use of the single-character control code (") causes characters to be stored as a full 8-bit ASCII code.

END OF PROGRAM

The special symbol "$" indicates the end of a program. When the Assembler encounters the "$" it terminates the PASS.

END OF TAPE

When several tapes are to be assembled together, each, except the last (which ends in "$"), should have as its last symbol the pseudo-instruction PAUSE. This causes the MACRO-8 Assembler to stop processing and halt the computer. After placing a new tape in the reader, assembly can be continued by depressing CONTINUE.

ALTERATIONS TO THE SYMBOL TABLE

There are two pseudo-instructions that may be used to alter the permanent symbol table (during PASS 1):

EXPUNGE — EXPUNGE the entire symbol table, except for the pseudo-instructions.

FIXTAB — FIX the symbol TABle. All symbols that are currently in the symbol table are fixed.

Example:

```
CLSF=6141
FIXTAB
```

would define CLSF as a permanent symbol.

```
EXPUNGE
TAD=1000
FIXTAB
```

would place the symbol TAD in the assembler's permanent symbol table. All other symbols would have been expunged.

CHAPTER 7

MACROS

When writing a program, it often happens that certain coding sequences are used several times with just the arguments changed. If so, it is convenient if the entire sequence can be generated by a single statement. To do this, the coding sequence is defined with dummy arguments as a macro. A single statement referring to the macro by name, along with a list of real arguments, will generate the correct sequence in line with the rest of the coding.

The macro name must be defined before it is used. The macro is defined by means of the pseudo-instruction DEFINE followed by the macro's name and a list of dummy arguments. For example:

A macro to move the contents of register A to register B and also leave the result in the accumulator, would be coded as follows:

```
DEFINE␣ MOVE ␣ DUMMY1 ␣ DUMMY2
<CLA
 TAD     DUMMY1
 DCA     DUMMY2
 TAD     DUMMY2>
```

The actual choice of symbols used as dummy arguments is arbitrary; however, they may not be defined or referenced prior to the macro definition.

The above definition of the macro MOVE is identical to the following:

```
DEFINE␣ MOVE ␣ ARG1␣ ARG2
<CLA;TAD    ARG1;    DCA ARG2;    TAD ARG2>
```

The actual definition of the macro is enclosed in angle brackets.

When a macro name is processed by the assembler, the real arguments will replace the dummy arguments. For example:

Assuming that the macro MOVE has been defined as above,

```
*400
A,0       0400     0000
B, -6     0401     7772
```

```
MOVE _ A, B     0402    7200
$               0403    1200
                0404    3201
                0405    1201
```

NOTE: A macro need not have any arguments: For example, a sequence of coding to rotate the C(AC) and C(L) six places to the left might be encoded as a macro by means of

```
DEFINE _ ROTL 6
<RTL; RTL; RTL>
```

The entire macro definition is placed in the Macro Table, two characters per word, with a dummy argument value replacing the symbolic name. Example:

```
DEFINE _ LOAD _ A
<CLA
 TAD A>
```

is stored, in the Macro Table, roughly as follows:

| CL | A) | TA | D_ | 7700 | >00 |

where the vertical lines indicate successive 12-bit words. Comments and line-feeds are not stored.

The macro definition can consist of any valid coding except for TEXT or " type statements.

RESTRICTIONS

1. Macros cannot be nested; i.e., another macro name or definition cannot appear in a macro definition and cannot be brought in as an argument at reference time.

2. TEXT or " type statements cannot appear in a macro definition.

3. Arguments cannot be:

   a. Macro name
   b. TEXT pseudo-instruction or " special character

4. The symbols used as dummy arguments must not have been previously defined or referenced.

5. A macro may not be redefined. Example:

```
DEFINE _ LOOP _ A _ B
<TAD     A
 DCA     B
 TAD     COUNT
 ISZ     B
 JMP     .-2>
```

The symbol "COUNT" is not a dummy argument but an actual symbol.

A macro is referenced by giving the macro name, a space, and then the list of real arguments, separated by commas. There must be at least as many arguments in the macro reference as in the corresponding macro definition. When a macro is referenced, its definition is found, expanded, and the real arguments replace the dummy arguments. The expanded macro is then processed in the normal fashion.

```
....
LOOP _ X, Y2
.....
```

is equivalent to:

```
....
TAD     X
DCA     Y2
TAD     COUNT
ISZ     Y2
JMP     .-2
....
```

NOTE: The macro table shares the available space (604_{10} registers) with the symbol table. Thus the programmer must be aware of the amount of room required by his macros and the fact that each symbol occupies four words of memory. Also, the arguments of a macro call are temporarily stored in this buffer space while the macro is being expanded.

CHAPTER 8

ERROR DIAGNOSTICS

The format of the error messages is:

ERROR CODE ADDRESS

Where ERROR CODE is a 2-character code which specifies the type of error, and ADDRESS is either the absolute octal address where the error occurred or the address of the error relative to the last symbolic tag (if there was one) on this page.

Assembly will continue or may be continued after all errors except SE (Symbol Table Exceeded). If an SE error occurs, the Assembler will halt and may not be restarted.

ERROR MESSAGES

PE Current, Non-Zero Page Exceeded

An attempt was made to

1. override a literal with an instruction or
2. override an instruction with a literal.

This can be corrected by

a. decreasing the number of literals on the page
b. decreasing the number of instructions on the page

ZE Zero Page Exceeded

Same as PE only with reference to page 0

ID Illegal Redefinition of a Symbol

An attempt was made to give a previously defined symbol a new value not via the "=". The symbol was not redefined. (This is similar to the Duplicate Tag diagnostic of PAL III).

IC Illegal Character

1. # % ' : ? @ \ were processed neither in a comment nor a TEXT field. The character is ignored and the assembly continued.

2. A non-valid character was processed. The computer will halt with the illegal character displayed in the accumulator. Assembly may be continued by putting the desired character in the SWITCH REGISTER and depressing CONTINUE.

IE Illegal Equals

An equal sign was used in the wrong context. Examples:

```
TAD    A + = B
A + B = C        (The expression to the left of the equal sign is not a single symbol)
```

II Illegal Indirect

An out of page reference was made, and a link could not be generated because the indirect bit was already set. Example:

```
*200
 TAD    I    A
  .
  .
  .
 PAGE
 A,    CMA    CLL
```

LG Link Generated

A link was generated for an out of page reference at this address. Example:

```
*200                          Generated Binary
 TAD    A                     0200       1777
   .  .  .
   .  .  .                    0377       0400
 PAGE
 A,    CMA,    CLL            0400       7140
```

SE Symbol Table Exceeded

The Symbol Table overlaps the Macro Table or vice versa. Assembly is halted and cannot be continued.

IM Illegal Format in a Macro Definition

The expression after the DEFINE pseudo-instruction does not comply with the macro definition, position, or structural rules. Example:

A macro name is referenced before the macro definition.

US Undefined Symbol

A symbol has been processed during PASS 2 that was not defined by the end of PASS 1.

MP Missing Parameter in a Macro Call

An argument, called for by the macro definition, is missing.

Example:

```
DEFINE    MAC    A    B

< TAD     A
  CIA
  DCA     B >

MAC       SUM
```

BE Two MACRO-8 internal tables have overlapped. This situation can usually be corrected by decreasing the number of current page literals used prior to this point on the page. If the error persists, please contact the Small Computer Systems Programming Group at Digital Equipment Corporation for assistance.

CHAPTER 9

OPERATING INSTRUCTIONS

MACRO-8 is a 2-pass assembler with an optional third pass which produces an octal/symbolic assembly listing. During the first pass, MACRO-8 processes the source tape and places all symbol definitions and macro definitions in its symbol table and macro table, respectively. During the second pass, MACRO-8 processes the source tape and punches the Binary Format Tape. At the end of PASS 2, MACRO-8 prints the Symbol Table (it is also punched if the 33-ASR PUNCH is turned on). This punched table may be read by DDT (See Digital-8-4-S). The third pass provides a listing of the generated octal code and the original source language.

There are two versions of MACRO-8 which differ with respect to their use of input/output equipment: the low speed version uses the 33-ASR Reader for all input and the 33-ASR Punch for all output; the high speed version uses the Type 750 Photoelectric Reader for all input, the Type 75 High Speed Punch for binary output, and the 33-ASR Punch for printable output such as error printouts, symbol table punching and listing, and third pass assembly listing.

> NOTE: In the high speed version of MACRO-8, the Type 75 Punch may be used as the printable output device by changing the contents of location 0004 from 2600 to 0600. This is useful for long third pass listings, since the punched output from the 75 Punch can be subsequently listed off-line. It is advised that this change not be made until pass 3, so that pass 1 and pass 2 error messages will come out on the 33-ASR.

1. Load MACRO-8 with the Binary Loader (See Digital-8-2-U).
2. Put the source tape in the reader.
3. Set the SWITCH REGISTER to 0200.
4. Depress LOAD ADDRESS.
5. Set switch options (See Table 9-1).
6. Depress START.
7. Turn on the 33-ASR reader (if using the low speed version).

8. When MACRO-8 stops reading (after processing a PAUSE statement), place the next tape in the reader and depress CONTINUE. Repeat this step until all tapes have been processed.

9. When MACRO-8 encounters the terminating character, dollar sign ($), it performs one of the following sets of events depending upon what pass has just been completed. Proper operator intervention is then required.

Pass Just Completed	Events	Operator Intervention
1	Set up for PASS 2	Turn on 33-ASR punch (in high speed MACRO-8, symbol table is output via 33-ASR). Put source tape in reader; hit CONTINUE to enter PASS 2.
2	Terminate current assembly; punch out page 0 constants, checksum and trailer code on binary tape; print and punch rubout, the alphanumerically ordered symbol table, an EOT code, a rubout, and trailer code; Set up for PASS 1.	(a) If PASS 3 is desired: (In the high speed version of MACRO-8, the contents of register 0004 could be altered at this point to change output devices). Go to step 2 of the operating instructions, making sure to set AC switch 3 up at step 5. (b) If PASS 3 not desired: Turn off 33-ASR punch, put next program to be assembled in the reader. Hit CONTINUE to enter PASS 1.
3	Terminate assembly listing: Set up for PASS 1.	Turn off 33-ASR punch; put next program to be assembled in the reader; hit CONTINUE to enter PASS 1.

TABLE 9-1 SWITCH OPTIONS

Switch Up	Result
None	MACRO-8 will enter the next pass as defined in the preceding table. For example: if the previous assembly was terminated during or at the end of PASS 1, restarting MACRO-8 with no switches up would cause PASS 2 to be entered. MACRO-8 initially starts at PASS 1.
0	Restore symbol table to only the permanent symbols and enter PASS 1.
1	Enter PASS 2.

TABLE 9-1 SWITCH OPTIONS (continued)

Switch Up	Result
2	Enter PASS 1 without erasing any previously defined symbols.
3	Enter PASS 3. During PASS 3, MACRO-8 outputs an octal/symbolic listing of the assembled program. If this pass is terminated before completion, either switch options 0 or 2 may be used to return to PASS 1 for subsequent assemblies. MACRO-8 will output as much of the source statement (symbolic) as its internal storage capacity will allow. Because of the internal operations during the processing of macro statements, the symbolic output may be meaningless.
10	The double precision integer and double precision floating point processors are deleted and may be used for storage of user defined symbols. This increases the size of the symbol table by 64_{10} symbols.
11	The macro processor and the number processors (above) are deleted and may be used for storage of user defined symbols. This increases the size of the symbol table by 125_{10} symbols.

NOTE: Switches 10 and 11 are sensed whenever PASS 1 is entered. MACRO-8 would have to be reloaded to handle subsequent programs that use macros, double precision integers, or floating point numbers.

The Binary Format Tape produced during PASS 2 may be loaded by the Binary Loader. When the loading is completed, the accumulator should contain zero which indicates that it has loaded correctly.

The PASS 3 output is of the following format:

AAAA NNNN (Symbolic) CR/LF

Where AAAA is the absolute octal address and NNNN is the generated code. Literals are somewhat out of phase with the octal. Example:

```
                    *200
0200        1377    TAD (1
0201        3776    DCA A
0376        4000    *4000
0377        0001
4000        0000    A, 0
```

SYMBOL TABLE MODIFICATION

Because of the small amount of core (604_{10} registers) remaining to be used for programmer symbols and the macro table, the following suggestions are offered which may allow a particular installation or individual to conserve on table space.

By use of the pseudo-ops EXPUNGE and FIXTAB, unnecessary instruction mnemonics can be removed from the symbol table thus making more space available for programmer defined symbols and macros. This also decreases assembly time as the never used instruction symbols are not involved in the symbol table searches. The most often used instruction mnemonics should be assembled first, so that they will be in core next to the special characters and pseudo-instructions. This is desirable because the symbol search routine starts searching at the top of the table and works down.

At an installation that does not have a piece of optional equipment available, the corresponding instruction set can be removed. A symbolic tape beginning with EXPUNGE, containing all necessary instruction mnemonics, and ending with FIXTAB and the $ sign could be assembled (only PASS 1 is needed) by MACRO-8 prior to any other assemblies. Example:

```
EXPUNGE
AND=0000
TAD=1000
CLA=7200
  .
  .
  .
FIXTAB
  $          (The pseudo-op PAUSE could also be used with this tape, the
             first of a multiple tape assembly.)
```

APPENDIX 1

MACRO-8 SYMBOL TABLE

```
/MEMORY REFERENCE INSTRUCTIONS
AND=0000
TAD=1000
ISZ=2000
DCA=3000
JMS=4000
JMP=5000
IOT=6000
OPR=7000
/MICROINSTRUCTIONS
NOP=7000
CLA=7200
CLL=7100
CMA=7040
CML=7020
RAR=7010
RTR=7012
RAL=7004
RTL=7006
IAC=7001
SMA=7500
SZA=7440
SPA=7510
SNA=7450
SNL=7420
SZL=7430
SKP=7410
OSR=7404
HLT=7402
/COMBINED MICROINSTRUCTIONS
CIA=7041
LAS=7604
STA=7240
STL=7120
GLK=7204
/PROGRAM INTERRUPT
ION=6001
IOF=6002
/ANALOG TO DIGITAL CONVERTER
ADC=6004
/HIGH SPEED PERFORATED TAPE READER
RSF=6011
RRB=6012
RFC=6014
/HIGH SPEED PERFORATED TAPE PUNCH
PSF=6021
PCF=6022
PPC=6024
PLS=6026
/TELETYPE KEYBOARD/READER
KSF=6031
KCC=6032
KRS=6034
KRB=6036
/TELETYPE TELEPRINTER/PUNCH
TSF=6041
TCF=6042
TPC=6044
TLS=6046
/FLOATING POINT INTERPRETIVE COMMANDS
FEXT=0000
FADD=1000
FSUB=2000
FMPY=3000
FDIV=4000
FGET=5000
FPUT=6000
FNOR=7000
/OSCILLOSCOPE AND PRECISION CRT
/DISPLAY
DCX=6051
DXL=6053
DCY=6061
DYL=6063
DIX=6054
DIY=6064
DXS=6057
DYS=6067
DSF=6071
DCF=6072
DLB=6074
/INCREMENTAL PLOTTER
PLSF=6501
PLCF=6502
PLPU=6504
PLPR=6511
PLDU=6512
PLDD=6514
PLUD=6522
PLPL=6521
PLPD=6524
/LINE PRINTER
LCF=6652
LPR=6655
LSF=6661
LCB=6662
LLD=6664
/CARD READER AND CONTROL
CRSF=6632
CERS=6634
CRRB=6671
CRSA=6672
```

```
CRSB=6674
/CARD PUNCH CONTROL
CPSF=6631
CPCF=6641
CPSE=6642
CPLB=6644
/AUTOMATIC MAGNETIC TAPE CONTROL
MSCR=6701
MCD=6702
MTS=6706
MSUR=6711
MNC=6712
MTC=6716
MSWF=6721
MDWF=6722
MCWF=6722
MEWF=6722
MIWF=6722
MSEF=6731
MDEF=6732
MCED=6732
MEEF=6732
MIEF=6732
MTRS=6734
MCC=6741
MRWC=6742
MRCA=6744
MCA=6745
/COMBINED INSTRUCTIONS
MMMM=6757
MMMF=6757
/AUTOMATIC MULTIPLY-DIVIDE
CAM=6101
LAR=6104
LMQ=6102
RDA=6112
MUL=6111
DIV=6121
SZO=6114
SAF=6124
RDM=6122
/MICROTAPE INSTRUCTIONS
MMLS=6751
MMLM=6752
MMLF=6754
MMSF=6761
MMCF=6772
MMSC=6771
MMRS=6774
MMCC=6762
MMLC=6766
MMML=6766
/MEMORY PARITY
SMP=6101
CMP=6102
/TYPE 138/139 ANALOG TO DIGITAL
/CONVERTER
ADSF=6531
ADCV=6532
ADRB=6534
ADCC=6541
ADSC=6542
ADIC=6544
/SERIAL MAGNETIC DRUM SYSTEM
DRCR=6603
DRCW=6605
DRCF=6611
DREF=6612
DRTS=6615
DRSE=6621
DRSC=6622
DRCN=6624
/EXTENDED ARITHMETIC ELEMENT
MUY=7405
DVI=7407
NMI=7411
SHL=7413
ASR=7415
LSR=7417
MQL=7421
SCA=7441
MQA=7501
/MAGNETIC TAPE SYSTEM
TIFM=6707
TSRD=6716
TSWR=6716
TSDF=6721
TSSR=6722
TSSI=6724
TSRS=6734
TWRT=6731
TCTI=6732
/MEMORY EXTENSION
RDF=6214
RIF=6224
RMF=6244
RIB=6234
CDF=6201
CIF=6202
```

APPENDIX 2

ASCII CHARACTER SET

These characters may be used in symbols.

Character	8-Bit From	6-Bit From	Character	8-Bit From	6-Bit From
A	301	01	S	323	23
B	302	02	T	324	24
C	303	03	U	325	25
D	304	04	V	326	26
E	305	05	W	327	27
F	306	06	X	330	30
G	307	07	Y	331	31
H	310	10	Z	332	32
I	311	11	0	260	60
J	312	12	1	261	61
K	313	13	2	262	62
L	314	14	3	263	63
M	315	15	4	264	64
N	316	16	5	265	65
O	317	17	6	266	66
P	320	20	7	267	67
Q	321	21	8	270	70
R	322	22	9	271	71

These characters are special.

Character	8-Bit From	6-Bit From	Meaning
!	241	41	Inclusive OR
"	242	42	Character pseudo-instruction
#	243	43	Illegal outside of (TEXT) or (") or comment
$	244	44	End of PASS
%	245	45	Illegal outside of (TEXT) or (") or comment
&	246	46	Logical AND
'	247	47	Illegal outside of (TEXT) or (") or comment
(	250	50	Define literal
)	251	51	Terminate literal
*	252	52	Set origin
+	253	53	2's complement addition
,	254	54	Define symbol
-	255	55	2's complement subtraction
.	256	56	Has value of CLC
/	257	57	Start of comment
:	272	72	Illegal outside of (TEXT) or (") or comment

Character	8-Bit From	6-Bit From	Meaning
;	273	73	Terminate expression
<	274	74	Start macro definition
=	275	75	Define Parameter
>	276	76	End macro definition
?	277	77	Illegal outside of (TEXT) or (") or comment
@	300	00	Illegal outside of (TEXT) or (") or comment
[	333	33	Define page 0 literal
\	334	34	Illegal outside of (TEXT) or (") or comment
]	335	35	End page 0 literal
↑	336	36	Illegal
←	337	37	Illegal
Line-feed	212		Used for formatting (ignored)
Return	215		Terminate line
Space	240		Address delimiter or inclusive OR
Rubout	377		Ignored
Form-feed	214		Ignored
Blank	000		Ignored
Code 200	200		Ignored

digital
EQUIPMENT
CORPORATION
MAYNARD, MASSACHUSETTS

PRINTED IN U.S.A.