CHAPTER SIX

Programming the Basic Computer

IN THIS CHAPTER

6-1 Introduction

A total computer system includes both *hardware* and *software*. Hardware consists of the physical components and all associated equipment. Software refers to the programs that are written for the computer. It is possible to be familiar with various aspects of computer software without being concerned with details of how the computer hardware operates. It is also possible to design parts of the hardware without a knowledge of its software capabilities. However, those concerned with computer architecture should have a knowledge of both hardware and software because the two branches influence each other.

Writing a program for a computer consists of specifying, directly or indirectly, a sequence of machine instructions. Machine instructions inside the computer form a binary pattern which is difficult, if not impossible, for people to work with and understand. It is preferable to write programs with the more familiar symbols of the alphanumeric character set. As a consequence, there is a need for translating user-oriented symbolic programs into binary programs recognized by the hardware.

A program written by a user may be either dependent or independent of the physical computer that runs his program. For example, a program written in standard Fortran is machine independent because most computers provide a translator program that converts the standard Fortran program to the binary code of the computer available in the particular installation. But the translator program itself is machine dependent because it must translate the Fortran program to the binary code recognized by the hardware of the particular computer used.

This chapter introduces some elementary programming concepts and shows their relation to the hardware representation of instructions. The first part presents the basic operation and structure of a program that translates a user's symbolic program into an equivalent binary program. The discussion emphasizes the important concepts of the translator rather than the details of actually producing the program itself. The usefulness of various machine instructions is then demonstrated by means of several basic programming examples.

instruction set

The instruction set of the basic computer, whose hardware organization was explored in Chap. 5, is used in this chapter to illustrate many of the techniques commonly used to program a computer. In this way it is possible to explore the relationship between a program and the hardware operations that execute the instructions.

The 25 instructions of the basic computer are repeated in Table 6-1 to provide an easy reference for the programming examples that follow. Each instruction is assigned a three-letter symbol to facilitate writing symbolic programs. The first seven instructions are memory-reference instructions and the other 18 are register-reference and input–output instructions. A memory-reference instruction has three parts: a mode bit, an operation code of three bits, and a 12-bit address. The first hexadecimal digit of a memory-reference instruction includes the mode bit and the operation code. The other three digits specify the address. In an indirect address instruction the mode bit is 1 and the first hexadecimal digit ranges in value from 8 to E. In a direct mode, the range is from 0 to 6. The other 18 instructions have a 16-bit operation code. The code for each instruction is listed as a four-digit hexadecimal number. The first digit of a register-reference instruction is always 7. The first digit of an input–output instruction is always F. The symbol m used in the description column denotes the effective address. The letter M refers to the memory word (operand) found at the effective address.

6-2 Machine Language

A program is a list of instructions or statements for directing the computer to perform a required data-processing task. There are various types of programming languages that one may *write* for a computer, but the computer can *execute* programs only when they are represented internally in binary form. Programs

TABLE 6-1 Computer Instructions

Symbol	Hexadecimal code	Description
AND	0 or 8	AND M to AC
ADD	1 or 9	Add M to AC, carry to E
LDA	2 or A	Load AC from M
STA	3 or B	Store AC in M
BUN	4 or C	Branch unconditionally to m
BSA	5 or D	Save return address in m and branch to $m + 1$
ISZ	6 or E	Increment M and skip if zero
CLA	7800	Clear AC
CLE	7400	Clear E
CMA	7200	Complement AC
CME	7100	Complement E
CIR	7080	Circulate right E and AC
CIL	7040	Circulate left E and AC
INC	7020	Increment AC,
SPA	7010	Skip if AC is positive
SNA	7008	Skip if AC is negative
SZA	7004	Skip if AC is zero
SZE	7002	Skip if E is zero
HLT	7001	Halt computer
INP	F800	Input information and clear flag
OUT	F400	Output information and clear flag
SKI	F200	Skip if input flag is on
SKO	F100	Skip if output flag is on
ION	F080	Turn interrupt on
IOF	F040	Turn interrupt off

written in any other language must be translated to the binary representation of instructions before they can be executed by the computer. Programs written for a computer may be in one of the following categories:

1. *Binary code*. This is a sequence of instructions and operands in binary that list the exact representation of instructions as they appear in computer memory.
2. *Octal or hexadecimal code*. This is an equivalent translation of the binary code to octal or hexadecimal representation.
3. *Symbolic code*. The user employs symbols (letters, numerals, or special characters) for the operation part, the address part, and other parts of the instruction code. Each symbolic instruction can be translated into one binary coded instruction. This translation is done by a special program called an *assembler*. Because an assembler translates the sym-

assembly language

bols, this type of symbolic program is referred to as an *assembly language* program.

4. *High-level programming languages*. These are special languages developed to reflect the procedures used in the solution of a problem rather than be concerned with the computer hardware behavior. An example of a high-level programming language is *Fortran*. It employs problem-oriented symbols and formats. The program is written in a sequence of statements in a form that people prefer to think in when solving a problem. However, each statement must be translated into a sequence of binary instructions before the program can be executed in a computer. The program that translates a high-level language program to binary is called a *compiler*.

machine language

Strictly speaking, a *machine language* program is a binary program of category 1. Because of the simple equivalency between binary and octal or hexadecimal representation, it is customary to refer to category 2 as machine language. Because of the one-to-one relationship between a symbolic instruction and its binary equivalent, an assembly language is considered to be a machine-level language.

We now use the basic computer to illustrate the relation between binary and assembly languages. Consider the binary program listed in Table 6-2. The first column gives the memory location (in binary) of each instruction or operand. The second column lists the binary content of these memory locations. (The *location* is the address of the memory word where the instruction is stored. It is important to differentiate it from the address part of the instruction itself.) The program can be stored in the indicated portion of memory, and then executed by the computer starting from address 0. The hardware of the computer will execute these instructions and perform the intended task. However, a person looking at this program will have a difficult time understanding what is to be achieved when this program is executed. Nevertheless, the computer hardware recognizes *only* this type of instruction code.

TABLE 6-2 Binary Program to Add Two Numbers

Location	Instruction code
0	0010 0000 0000 0100
1	0001 0000 0000 0101
10	0011 0000 0000 0110
11	0111 0000 0000 0001
100	0000 0000 0101 0011
101	1111 1111 1110 1001
110	0000 0000 0000 0000

TABLE 6-3 Hexadecimal Program to Add Two Numbers

Location	Instruction
000	2004
001	1005
002	3006
003	7001
004	0053
005	FFE9
006	0000

Writing 16 bits for each instruction is tedious because there are too many digits. We can reduce the number of digits per instruction if we write the octal equivalent of the binary code. This will require six digits per instruction. On the other hand, we can reduce each instruction to four digits if we write the

hexadecimal code

equivalent hexadecimal code as shown in Table 6-3. The hexadecimal representation is convenient to use; however, one must realize that each hexadecimal digit must be converted to its equivalent 4-bit number when the program is entered into the computer. The advantage of writing binary programs in equivalent octal or hexadecimal form should be evident from this example.

The program in Table 6-4 uses the symbolic names of instructions (listed in Table 6-1) instead of their binary or hexadecimal equivalent. The address parts of memory-reference instructions, as well as operands, remain in their hexadecimal value. Note that location 005 has a negative operand because the sign bit in the leftmost position is 1. The inclusion of a column for comments provides some means for explaining the function of each instruction. Symbolic programs are easier to handle, and as a consequence, it is preferable to write programs with symbols. These symbols can be converted to their binary code equivalent to produce the binary program.

We can go one step further and replace each hexadecimal address by a

TABLE 6-4 Program with Symbolic Operation Codes

Location	Instruction	Comments
000	LDA 004	Load first operand into *AC*
001	ADD 005	Add second operand to *AC*
002	STA 006	Store sum in location 006
003	HLT	Halt computer
004	0053	First operand
005	FFE9	Second operand (negative)
006	0000	Store sum here

TABLE 6-5 Assembly Language Program to Add Two Numbers

```
      ORG 0      /Origin of program is location 0
      LDA A      /Load operand from location A
      ADD B      /Add operand from location B
      STA C      /Store sum in location C
      HLT        /Halt computer
A,    DEC 83     /Decimal operand
B,    DEC -23    /Decimal operand
C,    DEC 0      /Sum stored in location C
      END        /End of symbolic program
```

symbolic address and each hexadecimal operand by a decimal operand. This is convenient because one usually does not know exactly the numeric memory location of operands while writing a program. If the operands are placed in memory following the instructions, and if the length of the program is not known in advance, the numerical location of operands is not known until the end of the program is reached. In addition, decimal numbers are more familiar than their hexadecimal equivalents.

The program in Table 6-5 is the assembly-language program for adding two numbers. The symbol ORG followed by a number is not a machine instruction. Its purpose is to specify an *origin*, that is, the memory location of the next instruction below it. The next three lines have symbolic addresses. Their value is specified by their being present as a label in the first column. Decimal operands are specified following the symbol DEC. The numbers may be positive or negative, but if negative, they must be converted to binary in the signed-2's complement representation. The last line has the symbol END indicating the end of the program. The symbols ORG, DEC, and END, called *pseudoinstructions*, are defined in the next section. Note that all comments are preceded by a slash.

The equivalent Fortran program for adding two integer numbers is listed in Table 6-6. The two values for *A* and *B* may be specified by an input statement or by a data statement. The arithmetic operation for the two numbers is specified by one simple statement. The translation of this Fortran program into a binary program consists of assigning three memory locations, one each for the augend, addend, and sum, and then deriving the sequence of binary

TABLE 6-6 Fortran Program to Add Two Numbers

```
INTEGER A, B, C
DATA A,83  B,-23
C = A + B
END
```

instructions that form the sum. Thus a compiler program translates the symbols of the Fortran program into the binary values listed in the program of Table 6-2.

6-3 Assembly Language

A programming language is defined by a set of rules. Users must conform with all format rules of the language if they want their programs to be translated correctly. Almost every commercial computer has its own particular assembly language. The rules for writing assembly language programs are documented and published in manuals which are usually available from the computer manufacturer.

The basic unit of an assembly language program is a line of code. The specific language is defined by a set of rules that specify the symbols that can be used and how they may be combined to form a line of code. We will now formulate the rules of an assembly language for writing symbolic programs for the basic computer.

Rules of the Language

Each line of an assembly language program is arranged in three columns called fields. The fields specify the following information.

1. The *label* field may be empty or it may specify a symbolic address.
2. The *instruction* field specifies a machine instruction or a pseudoinstruction.
3. The *comment* field may be empty or it may include a comment.

symbolic address

A symbolic address consists of one, two, or three, but not more than three alphanumeric characters. The first character must be a letter; the next two may be letters or numerals. The symbol can be chosen arbitrarily by the programmer. A symbolic address in the label field is terminated by a comma so that it will be recognized as a label by the assembler.

The instruction field in an assembly language program may specify one of the following items:

1. A memory-reference instruction (MRI)
2. A register-reference or input–output instruction (non-MRI)
3. A pseudoinstruction with or without an operand

A memory-reference instruction occupies two or three symbols separated by spaces. The first must be a three-letter symbol defining an MRI operation code from Table 6-1. The second is a symbolic address. The third symbol, which may or may not be present, is the letter I. If I is missing, the line denotes a direct address instruction. The presence of the symbol I denotes an indirect address instruction.

A non-MRI is defined as an instruction that does not have an address part. A non-MRI is recognized in the instruction field of a program by any one of the three-letter symbols listed in Table 6-1 for the register-reference and input–output instructions.

The following is an illustration of the symbols that may be placed in the instruction field of a program.

```
CLA          non-MRI
ADD OPR      direct address MRI
ADD PTR I    indirect address MRI
```

The first three-letter symbol in each line must be one of the instruction symbols of the computer and must be listed in Table 6-1. A memory-reference instruction, such as ADD, must be followed by a symbolic address. The letter I may or may not be present.

A symbolic address in the instruction field specifies the memory location of an operand. This location must be defined somewhere in the program by appearing again as a label in the first column. To be able to translate an assembly language program to a binary program, it is absolutely necessary that each symbolic address that is mentioned in the instruction field *must* occur again in the label field.

pseudoinstruction

A pseudoinstruction is not a machine instruction but rather an instruction to the assembler giving information about some phase of the translation. Four pseudoinstructions that are recognized by the assembler are listed in Table 6-7. (Other assembly language programs recognize many more pseudoinstructions.) The ORG (origin) pseudoinstruction informs the assembler that the instruction or operand in the following line is to be placed in a memory location specified by the number next to ORG. It is possible to use ORG more than once in a program to specify more than one segment of memory. The END symbol

TABLE 6-7 Definition of Pseudoinstructions

Symbol	Information for the Assembler
ORG N	Hexadecimal number N is the memory location for the instruction or operand listed in the following line
END	Denotes the end of symbolic program
DEC N	Signed decimal number N to be converted to binary
HEX N	Hexadecimal number N to be converted to binary

is placed at the end of the program to inform the assembler that the program is terminated. The other two pseudoinstructions specify the radix of the operand and tell the assembler how to convert the listed number to a binary number.

The third field in a program is reserved for comments. A line of code may or may not have a comment, but if it has, it must be preceded by a slash for the assembler to recognize the beginning of a comment field. Comments are useful for explaining the program and are helpful in understanding the step-by-step procedure taken by the program. Comments are inserted for explanation purposes only and are neglected during the binary translation process.

An Example

The program of Table 6-8 is an example of an assembly language program. The first line has the pseudoinstruction ORG to define the origin of the program at memory location $(100)_{16}$. The next six lines define machine instructions, and the last four have pseudoinstructions. Three symbolic addresses have been used and each is listed in column 1 as a label and in column 2 as an address of a memory-reference instruction. Three of the pseudoinstructions specify operands, and the last one signifies the END of the program.

When the program is translated into binary code and executed by the computer it will perform a subtraction between two numbers. The subtraction is performed by adding the minuend to the 2's complement of the subtrahend. The subtrahend is a negative number. It is converted into a binary number in signed-2's complement representation because we dictate that all negative numbers be in their 2's complement form. When the 2's complement of the subtrahend is taken (by complementing and incrementing the *AC*), −23 converts to +23 and the difference is 83 + (2's complement of −23) = 83 + 23 = 106.

TABLE 6-8 Assembly Language Program to Subtract Two Numbers

```
        ORG 100     /Origin of program is location 100
        LDA SUB     /Load subtrahend to AC
        CMA         /Complement AC
        INC         /Increment AC
        ADD MIN     /Add minuend to AC
        STA DIF     /Store difference
        HLT         /Halt computer
MIN,    DEC 83      /Minuend
SUB,    DEC -23     /Subtrahend
DIF,    HEX 0       /Difference stored here
        END         /End of symbolic program
```

Translation to Binary

assembler

The translation of the symbolic program into binary is done by a special program called an *assembler*. The tasks performed by the assembler will be better understood if we first perform the translation on paper. The translation of the symbolic program of Table 6-8 into an equivalent binary code may be done by scanning the program and replacing the symbols by their machine code binary equivalent. Starting from the first line, we encounter an ORG pseudoinstruction. This tells us to start the binary program from hexadecimal location 100. The second line has two symbols. It must be a memory-reference instruction to be placed in location 100. Since the letter I is missing, the first bit of the instruction code must be 0. The symbolic name of the operation is LDA. Checking Table 6-1 we find that the first hexadecimal digit of the instruction should be 2. The binary value of the address part must be obtained from the address symbol SUB. We scan the label column and find this symbol in line 9. To determine its hexadecimal value we note that line 2 contains an instruction for location 100 and every other line specifies a machine instruction or an operand for sequential memory locations. Counting lines, we find that label SUB in line 9 corresponds to memory location 107. So the hexadecimal address of the instruction LDA must be 107. When the two parts of the instruction are assembled, we obtain the hexadecimal code 2107. The other lines representing machine instructions are translated in a similar fashion and their hexadecimal code is listed in Table 6-9.

Two lines in the symbolic program specify decimal operands with the pseudoinstruction DEC. A third specifies a zero by means of a HEX pseudoinstruction (DEC could be used as well). Decimal 83 is converted to binary and placed in location 106 in its hexadecimal equivalent. Decimal −23 is a negative number and must be converted into binary in signed-2's complement form.

TABLE 6-9 Listing of Translated Program of Table 6-8

Hexadecimal code			
Location	Content	Symbolic program	
			ORG 100
100	2107		LDA SUB
101	7200		CMA
102	7020		INC
103	1106		ADD MIN
104	3108		STA DIF
105	7001		HLT
106	0053	MIN,	DEC 83
107	FFE9	SUB,	DEC −23
108	0000	DIF,	HEX 0
			END

The hexadecimal equivalent of the binary number is placed in location 107. The END symbol signals the end of the symbolic program telling us that there are no more lines to translate.

address symbol table

The translation process can be simplified if we scan the entire symbolic program twice. No translation is done during the first scan. We merely assign a memory location to each machine instruction and operand. The location assignment will define the address value of labels and facilitate the translation process during the second scan. Thus in Table 6-9, we assign location 100 to the first instruction after ORG. We then assign sequential locations for each line of code that has a machine instruction or operand up to the end of the program. (ORG and END are not assigned a numerical location because they do not represent an instruction or an operand.) When the first scan is completed, we associate with each label its location number and form a table that defines the hexadecimal value of each symbolic address. For this program, the address symbol table is as follows:

Address symbol	Hexadecimal address
MIN	106
SUB	107
DIF	108

During the second scan of the symbolic program we refer to the address symbol table to determine the address value of a memory-reference instruction. For example, the line of code LDA SUB is translated during the second scan by getting the hexadecimal value of LDA from Table 6-1 and the hexadecimal value of SUB from the address-symbol table listed above. We then assemble the two parts into a four-digit hexadecimal instruction. The hexadecimal code can be easily converted to binary if we wish to know exactly how this program resides in computer memory.

When the translation from symbols to binary is done by an assembler program, the first scan is called the *first pass*, and the second is called the *second pass*.

6-4 The Assembler

An assembler is a program that accepts a symbolic language program and produces its binary machine language equivalent. The input symbolic program is called the *source program* and the resulting binary program is called the *object program*. The assembler is a program that operates on character strings and produces an equivalent binary interpretation.

Representation of Symbolic Program in Memory

Prior to starting the assembly process, the symbolic program must be stored in memory. The user types the symbolic program on a terminal. A loader program is used to input the characters of the symbolic program into memory. Since the program consists of symbols, its representation in memory must use an alphanumeric character code. In the basic computer, each character is represented by an 8-bit code. The high-order bit is always 0 and the other seven bits are as specified by ASCII. The hexadecimal equivalent of the character set is listed in Table 6-10. Each character is assigned two hexadecimal digits which can be easily converted to their equivalent 8-bit code. The last entry in the table does not print a character but is associated with the physical movement of the cursor in the terminal. The code for CR is produced when the return key is depressed. This causes the "carriage" to return to its initial position to start typing a new line. The assembler recognizes a CR code as the end of a line of code.

line of code

A line of code is stored in consecutive memory locations with two characters in each location. Two characters can be stored in each word since a memory word has a capacity of 16 bits. A label symbol is terminated with a comma. Operation and address symbols are terminated with a space and the end of the line is recognized by the CR code. For example, the following line of code:

```
PL3,    LDA SUB I
```

TABLE 6-10 Hexadecimal Character Code

Character	Code	Character	Code	Character	Code	
A	41	Q	51	6	36	
B	42	R	52	7	37	
C	43	S	53	8	38	
D	44	T	54	9	39	
E	45	U	55	space	20	
F	46	V	56	(	28	
G	47	W	57	)	29	
H	48	X	58	*	2A	
I	49	Y	59	+	2B	
J	4A	Z	5A	,	2C	
K	4B	0	30	−	2D	
L	4C	1	31	.	2E	
M	4D	2	32	/	2F	
N	4E	3	33	=	3D	
O	4F	4	34	CR	0D	(carriage
P	50	5	35			return)

TABLE 6-11 Computer Representation of the Line of Code: PL3, LDA SUB I

Memory word	Symbol	Hexadecimal code	Binary representation
1	P L	50 4C	0101 0000 0100 1100
2	3 ,	33 2C	0011 0011 0010 1100
3	L D	4C 44	0100 1100 0100 0100
4	A	41 20	0100 0001 0010 0000
5	S U	53 55	0101 0011 0101 0101
6	B	42 20	0100 0010 0010 0000
7	I CR	49 0D	0100 1001 0000 1101

is stored in seven consecutive memory locations, as shown in Table 6-11. The label PL3 occupies two words and is terminated by the code for comma (2C). The instruction field in the line of code may have one or more symbols. Each symbol is terminated by the code for space (20) except for the last symbol, which is terminated by the code of carriage return (0D). If the line of code has a comment, the assembler recognizes it by the code for a slash (2F). The assembler neglects all characters in the comment field and keeps checking for a CR code. When this code is encountered, it replaces the space code after the last symbol in the line of code.

The input for the assembler program is the user's symbolic language program in ASCII. This input is scanned by the assembler twice to produce the equivalent binary program. The binary program constitutes the output generated by the assembler. We will now describe briefly the major tasks that must be performed by the assembler during the translation process.

First Pass

A two-pass assembler scans the entire symbolic program twice. During the first pass, it generates a table that correlates all user-defined address symbols with their binary equivalent value. The binary translation is done during the second pass. To keep track of the location of instructions, the assembler uses a memory word called a *location counter* (abbreviated LC). The content of LC stores the value of the memory location assigned to the instruction or operand presently being processed. The ORG pseudoinstruction initializes the location counter to the value of the first location. Since instructions are stored in sequential locations, the content of LC is incremented by 1 after processing each line of code. To avoid ambiguity in case ORG is missing, the assembler sets the location counter to 0 initially.

location counter (LC)

The tasks performed by the assembler during the first pass are described in the flowchart of Fig. 6-1. LC is initially set to 0. A line of symbolic code is analyzed to determine if it has a label (by the presence of a comma). If the line

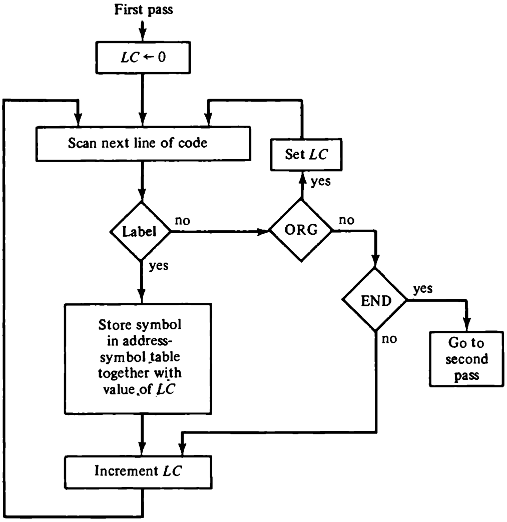


Figure 6-1 Flowchart for first pass of assembler.

of code has no label, the assembler checks the symbol in the instruction field. If it contains an ORG pseudoinstruction, the assembler sets LC to the number that follows ORG and goes back to process the next line. If the line has an END pseudoinstruction, the assembler terminates the first pass and goes to the second pass. (Note that a line with ORG or END should not have a label.) If the line of code contains a label, it is stored in the address symbol table together with its binary equivalent number specified by the content of LC. Nothing is stored in the table if no label is encountered. LC is then incremented by 1 and a new line of code is processed.

For the program of Table 6-8, the assembler generates the address symbol table listed in Table 6-12. Each label symbol is stored in two memory locations and is terminated by a comma. If the label contains less than three characters, the memory locations are filled with the code for space. The value found in LC while the line was processed is stored in the next sequential memory location. The program has three symbolic addresses: MIN, SUB, and DIF. These symbols represent 12-bit addresses equivalent to hexadecimal 106, 107, and 108,

TABLE 6-12 Address Symbol Table for Program in Table 6-8

Memory word	Symbol or (LC)*	Hexadecimal code	Binary representation
1	M I	4D 49	0100 1101 0100 1001
2	N ,	4E 2C	0100 1110 0010 1100
3	(LC)	01 06	0000 0001 0000 0110
4	S U	53 55	0101 0011 0101 0101
5	B ,	42 2C	0100 0010 0010 1100
6	(LC)	01 07	0000 0001 0000 0111
7	D I	44 49	0100 0100 0100 1001
8	F ,	46 2C	0100 0110 0010 1100
9	(LC)	01 08	0000 0001 0000 1000

* (LC) designates content of location counter.

respectively. The address symbol table occupies three words for each label symbol encountered and constitutes the output data that the assembler generates during the first pass.

Second Pass

table-lookup

Machine instructions are translated during the second pass by means of table-lookup procedures. A table-lookup procedure is a search of table entries to determine whether a specific item matches one of the items stored in the table. The assembler uses four tables. Any symbol that is encountered in the program must be available as an entry in one of these tables; otherwise, the symbol cannot be interpreted. We assign the following names to the four tables:

1. Pseudoinstruction table.
2. MRI table.
3. Non-MRI table.
4. Address symbol table.

The entries of the pseudoinstruction table are the four symbols ORG, END, DEC, and HEX. Each entry refers the assembler to a subroutine that processes the pseudoinstruction when encountered in the program. The MRI table contains the seven symbols of the memory-reference instructions and their 3-bit operation code equivalent. The non-MRI table contains the symbols for the 18 register-reference and input–output instructions and their 16-bit binary code equivalent. The address symbol table is generated during the first pass of the assembly process. The assembler searches these tables to find the symbol that it is currently processing in order to determine its binary value.

The tasks performed by the assembler during the second pass are described in the flowchart of Fig. 6-2. LC is initially set to 0. Lines of code are then analyzed one at a time. Labels are neglected during the second pass, so the assembler goes immediately to the instruction field and proceeds to check the first symbol encountered. It first checks the pseudoinstruction table. A match with ORG sends the assembler to a subroutine that sets LC to an initial value. A match with END terminates the translation process. An operand pseudoinstruction causes a conversion of the operand into binary. This operand is placed in the memory location specified by the content of LC. The location counter is then incremented by 1 and the assembler continues to analyze the next line of code.

If the symbol encountered is not a pseudoinstruction, the assembler refers to the MRI table. If the symbol is not found in this table, the assembler refers to the non-MRI table. A symbol found in the non-MRI table corresponds to a register reference or input–output instruction. The assembler stores the 16-bit instruction code into the memory word specified by LC. The location counter is incremented and a new line analyzed.

When a symbol is found in the MRI table, the assembler extracts its equivalent 3-bit code and inserts it in bits 2 through 4 of a word. A memory reference instruction is specified by two or three symbols. The second symbol is a symbolic address and the third, which may or may not be present, is the letter I. The symbolic address is converted to binary by searching the address symbol table. The first bit of the instruction is set to 0 or 1, depending on whether the letter I is absent or present. The three parts of the binary instruction code are assembled and then stored in the memory location specified by the content of LC. The location counter is incremented and the assembler continues to process the next line.

error diagnostics

One important task of an assembler is to check for possible errors in the symbolic program. This is called *error diagnostics*. One such error may be an invalid machine code symbol which is detected by its being absent in the MRI and non-MRI tables. The assembler cannot translate such a symbol because it does not know its binary equivalent value. In such a case, the assembler prints an error message to inform the programmer that his symbolic program has an error at a specific line of code. Another possible error may occur if the program has a symbolic address that did not appear also as a label. The assembler cannot translate the line of code properly because the binary equivalent of the symbol will not be found in the address symbol table generated during the first pass. Other errors may occur and a practical assembler should detect all such errors and print an error message for each.

It should be emphasized that a practical assembler is much more complicated than the one explained here. Most computers give the programmer more flexibility in writing assembly language programs. For example, the user may be allowed to use either a number or a symbol to specify an address. Many assemblers allow the user to specify an address by an arithmetic expression. Many more pseudoinstructions may be specified to facilitate the programming

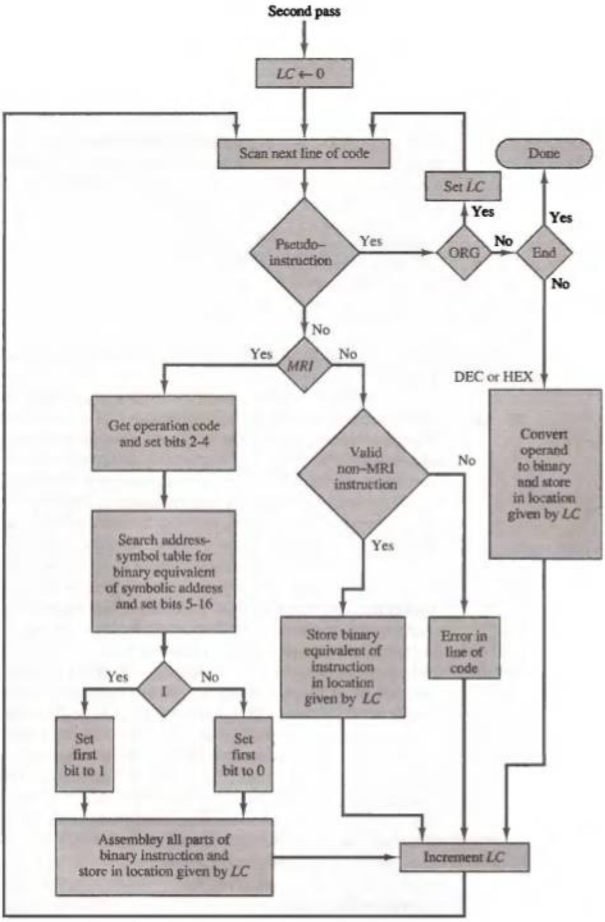

Second pass
$LC \leftarrow 0$
Scan next line of code
Set LC
Done
Yes
Yes
Pseudo-instruction
Yes
ORG
No
End
No
No
Yes
MRI
No
DEC or HEX
Get operation code and set bits 2-4
Valid non-MRI instruction
No
Convert operand to binary and store in location given by LC
Search address-symbol table for binary equivalent of symbolic address and set bits 5-16
Yes
Store binary equivalent of instruction in location given by LC
Error in line of code
Yes
I
No
Set first bit to 1
Set first bit to 0
Assembley all parts of binary instruction and store in location given by LC
Increment LC

Figure 6-2 Flowchart for second pass of assembler.

task. As the assembly language becomes more sophisticated, the assembler becomes more complicated.

6-5 Program Loops

A program loop is a sequence of instructions that are executed many times, each time with a different set of data. Program loops are specified in Fortran by a DO statement. The following is an example of a Fortran program that forms the sum of 100 integer numbers.

```
      DIMENSION  A(100)
      INTEGER  SUM,  A
      SUM = 0
      DO  3  J = 1,  100
   3  SUM = SUM + A(J)
```

Statement number 3 is executed 100 times, each time with a different operand A(J) for J = 1, 2, . . . , 100.

A system program that translates a program written in a high-level programming language such as the above to a machine language program is called a *compiler*. A compiler is a more complicated program than an assembler and requires knowledge of systems programming to fully understand its operation. Nevertheless, we can demonstrate the basic functions of a compiler by going through the process of translating the program above to an assembly language program. A compiler may use an assembly language as an intermediate step in the translation or may translate the program directly to binary.

compiler

The first statement in the Fortran program is a DIMENSION statement. This statement instructs the compiler to reserve 100 words of memory for 100 operands. The value of the operands is determined from an input statement (not listed in the program). The second statement informs the compiler that the numbers are integers. If they were of the *real* type, the compiler would have to reserve locations for floating-point numbers and generate instructions that perform the subsequent arithmetic with floating-point data. These two statements are nonexecutable and are similar to the pseudoinstructions in an assembly language. Suppose that the compiler reserves locations $(150)_{16}$ to $(1B3)_{16}$ for the 100 operands. These reserved memory words are listed in lines 19 to 118 in the translated program of Table 6-13. This is done by the ORG pseudoinstruction in line 18, which specifies the origin of the operands. The first and last operands are listed with a specific decimal number, although these values are not known during compilation. The compiler just reserves the data space in memory and the values are inserted later when an input data statement is executed. The line numbers in the symbolic program are for reference only and are not part of the translated symbolic program.

The indexing of the DO statement is translated into the instructions in

TABLE 6-13 Symbolic Program to Add 100 Numbers

Line			
1		ORG 100	/Origin of program is HEX 100
2		LDA ADS	/Load first address of operands
3		STA PTR	/Store in pointer
4		LDA NBR	/Load minus 100
5		STA CTR	/Store in counter
6		CLA	/Clear accumulator
7	LOP,	ADD PTR I	/Add an operand to *AC*
8		ISZ PTR	/Increment pointer
9		ISZ CTR	/Increment counter
10		BUN LOP	/Repeat loop again
11		STA SUM	/Store sum
12		HLT	/Halt
13	ADS,	HEX 150	/First address of operands
14	PTR,	HEX 0	/This location reserved for a pointer
15	NBR,	DEC −100	/Constant to initialized counter
16	CTR,	HEX 0	/This location reserved for a counter
17	SUM,	HEX 0	/Sum is stored here
18		ORG 150	/Origin of operands is HEX 150
19		DEC 75	/First operand
•			
•			
•			
118		DEC 23	/Last operand
119		END	/End of symbolic program

lines 2 through 5 and the constants in lines 13 through 16. The address of the first operand (150) is stored in location ADS in line 13. The number of times that Fortran statement number 3 must be executed is 100. So −100 is stored in location NBR. The compiler then generates the instructions in lines 2 through 5 to initialize the program loop. The address of the first operand is transferred to location PTR. This corresponds to setting A(J) to A(1). The number −100 is then transferred to location CTR. This location acts as a counter with its content incremented by one every time the program loop is executed. When the value of the counter reaches zero, the 100 operations will be completed and the program will exit from the loop.

Some compilers will translate the statement SUM = 0 into a machine instruction that initializes location SUM to zero. A reference to this location is then made every time Fortran statement number 3 is executed. A more intelligent compiler will realize that the sum can be formed in the accumulator and only the final result stored in location SUM. This compiler will produce an instruction in line 6 to clear the *AC*. It will also reserve a memory location symbolized by SUM (in line 17) for storing the value of this variable at the termination of the loop.

The program loop specified by the DO statement is translated to the sequence of instructions listed in lines 7 through 10. Line 7 specifies an indirect ADD instruction because it has the symbol I. The address of the current operand is stored in location PTR. When this location is addressed indirectly the computer takes the content of PTR to be the address of the operand. As a result, the operand in location 150 is added to the accumulator. Location PTR is then incremented with the ISZ instruction in line 8, so its value changes to the value of the address of the next sequential operand. Location CTR is incremented in line 9, and if it is not zero, the computer does not skip the next instruction. The next instruction is a branch (BUN) instruction to the beginning of the loop, so the computer returns to repeat the loop once again. When location CTR reaches zero (after the loop is executed 100 times), the next instruction is skipped and the computer executes the instructions in lines 11 and 12. The sum formed in the accumulator is stored in SUM and the computer halts. The halt instruction is inserted here for clarity; actually, the program will branch to a location where it will continue to execute the rest of the program or branch to the beginning of another program. Note that ISZ in line 8 is used merely to add 1 to the address pointer PTR. Since the address is a positive number, a skip will never occur.

pointer
counter

The program of Table 6-13 introduces the idea of a pointer and a counter which can be used, together with the indirect address operation, to form a program loop. The pointer points to the address of the current operand and the counter counts the number of times that the program loop is executed. In this example we use two memory locations for these functions. In computers with more than one processor register, it is possible to use one processor register as a pointer, another as a counter, and a third as an accumulator. When processor registers are used as pointers and counters they are called *index registers*. Index registers are discussed in Sec. 8-5.

6-6 Programming Arithmetic and Logic Operations

The number of instructions available in a computer may be a few hundred in a large system or a few dozen in a small one. Some computers perform a given operation with one machine instruction; others may require a large number of machine instructions to perform the same operation. As an illustration, consider the four basic arithmetic operations. Some computers have machine instructions to add, subtract, multiply, and divide. Others, such as the basic computer, have only one arithmetic instruction, such as ADD. Operations not included in the set of machine instructions must be implemented by a program.

We have shown in Table 6-8 a program for subtracting two numbers. Programs for the other arithmetic operations can be developed in a similar fashion.

Operations that are implemented in a computer with one machine instruction are said to be implemented by hardware. Operations implemented by a set of instructions that constitute a program are said to be implemented by software. Some computers provide an extensive set of hardware instructions designed to speed up common tasks. Others contain a smaller set of hardware instructions and depend more heavily on the software implementation of many operations. Hardware implementation is more costly because of the additional circuits needed to implement the operation. Software implementation results in long programs both in number of instructions and in execution time.

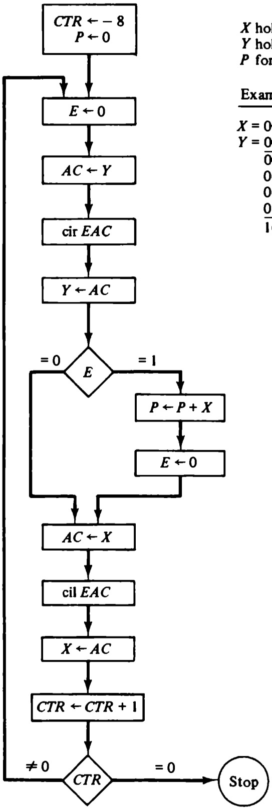
This section demonstrates the software implementation of a few arithmetic and logic operations. Programs can be developed for any arithmetic operation and not only for fixed-point binary data but for decimal and floating-point data as well. The hardware implementation of arithmetic operations is carried out in Chap. 10.

Multiplication Program

We now develop a program for multiplying two numbers. To simplify the program, we neglect the sign bit and assume positive numbers. We also assume that the two binary numbers have no more than eight significant bits so their product cannot exceed the word capacity of 16 bits. It is possible to modify the program to take care of the signs or use 16-bit numbers. However, the product may be up to 31 bits in length and will occupy two words of memory.

The program for multiplying two numbers is based on the procedure we use to multiply numbers with paper and pencil. As shown in the numerical example of Fig. 6-3, the multiplication process consists of checking the bits of the multiplier Y and adding the multiplicand X as many times as there are 1's in Y, provided that the value of X is shifted left from one line to the next. Since the computer can add only two numbers at a time, we reserve a memory location, denoted by P, to store intermediate sums. The intermediate sums are called partial products since they hold a partial product until all numbers are added. As shown in the numerical example under P, the partial product starts with zero. The multiplicand X is added to the content of P for each bit of the multiplier Y that is 1. The value of X is shifted left after checking each bit of the multiplier. The final value in P forms the product. The numerical example has numbers with four significant bits. When multiplied, the product contains eight significant bits. The computer can use numbers with eight significant bits to produce a product of up to 16 bits.

The flowchart of Fig. 6-3 shows the step-by-step procedure for program-



X holds the multiplicand
Y holds the multiplier
P forms the product

Example with four significant digits

	P
X = 0000 1111	
Y = 0000 1011	0000 0000
0000 1111	0000 1111
0001 1110	0010 1101
0000 0000	0010 1101
0111 1000	1010 0101
1010 0101	

Figure 6-3 Flowchart for multiplication program.

ming the multiplication operation. The program has a loop that is traversed eight times, once for each significant bit of the multiplier. Initially, location X holds the multiplicand and location Y holds the multiplier. A counter CTR is set to −8 and location P is cleared to zero.

The multiplier bit can be checked if it is transferred to the *E* register. This is done by clearing *E*, loading the value of Y into the *AC*, circulating right *E* and *AC* and storing the shifted number back into location Y. This bit stored in *E* is the low-order bit of the multiplier. We now check the value of *E*. If it is 1, the multiplicand X is added to the partial product P. If it is 0, the partial product does not change. We then shift the value of X once to the left by loading it into the *AC* and circulating left *E* and *AC*. The loop is repeated eight times by incrementing location CTR and checking when it reaches zero. When the counter reaches zero, the program exits from the loop with the product stored in location P.

The program in Table 6-14 lists the instructions for multiplying two unsigned numbers. The initialization is not listed but should be included when the program is loaded into the computer. The initialization consists of bringing the multiplicand and multiplier into locations X and Y, respectively; initializing the counter to −8; and initializing location P to zero. If these locations are not

TABLE 6-14 Program to Multiply Two Positive Numbers

```
        ORG 100
LOP,    CLE         /Clear E
        LDA Y       /Load multiplier
        CIR         /Transfer multiplier bit to E
        STA Y       /Store shifted multiplier
        SZE         /Check if bit is zero
        BUN ONE     /Bit is one; go to ONE
        BUN ZRO     /Bit is zero; go to ZRO
ONE,    LDA X       /Load multiplicand
        ADD P       /Add to partial product
        STA P       /Store partial product
        CLE         /Clear E
ZRO,    LDA X       /Load multiplicand
        CIL         /Shift left
        STA X       /Store shifted multiplicand
        ISZ CTR     /Increment counter
        BUN LOP     /Counter not zero; repeat loop
        HLT         /Counter is zero; halt
CTR,    DEC -8      /This location serves as a counter
X,      HEX 000F    /Multiplicand stored here
Y,      HEX 000B    /Multiplier stored here
P,      HEX 0       /Product formed here
        END
```

initialized, the program may run with incorrect data. The program itself is straightforward and follows the steps listed in the flowchart. The comments may help in following the step-by-step procedure.

This example has shown that if a computer does not have a machine instruction for a required operation, the operation can be programmed by a sequence of machine instructions. Thus we have demonstrated the software implementation of the multiplication operation. The corresponding hardware implementation is presented in Sec. 10-3.

Double-Precision Addition

When two 16-bit unsigned numbers are multiplied, the result is a 32-bit product that must be stored in two memory words. A number stored in two memory words is said to have double precision. When a partial product is computed, it is necessary that a double-precision number be added to the shifted multiplicand, which is also a double-precision number. For greater accuracy, the programmer may wish to employ double-precision numbers and perform arithmetic with operands that occupy two memory words. We now develop a program that adds two double-precision numbers.

One of the double-precision numbers is placed in two consecutive memory locations, AL and AH, with AL holding the 16 low-order bits. The other number is placed in BL and BH. The program is listed in Table 6-15. The two low-order portions are added and the carry transferred into *E*. The *AC* is cleared and the bit in *E* is circulated into the least significant position of the *AC*. The two high-order portions are then added to the carry and the double-precision sum is stored in CL and CH.

TABLE 6-15 Program to Add Two Double-Precision Numbers

```
        LDA AL      /Load A low
        ADD BL      /Add B low, carry in E
        STA CL      /Store in C low
        CLA         /Clear AC
        CIL         /Circulate to bring carry into AC(16)
        ADD AH      /Add A high and carry
        ADD BH      /Add B high
        STA CH      /Store in C high
        HLT
AL,     —           /Location of operands
AH,     —
BL,     —
BH,     —
CL,     —
CH,     —
```

Logic Operations

The basic computer has three machine instructions that perform logic operations: AND, CMA, and CLA. The LDA instruction may be considered as a logic operation that transfers a logic operand into the *AC*. In Sec. 4-5 we listed 16 different logic operations. All 16 logic operations can be implemented by software means because any logic function can be implemented using the AND and complement operations. For example, the OR operation is not available as a machine instruction in the basic computer. From DeMorgan's theorem we recognize the relation $x + y = (x'y')'$. The second expression contains only AND and complement operations. A program that forms the OR operation of two logic operands *A* and *B* is as follows:

```
LDA A     Load first operand A
CMA       Complement to get Ā
STA TMP   Store in a temporary location
LDA B     Load second operand B
CMA       Complement to get B̄
AND TMP   AND with Ā to get Ā ∧ B̄
CMA       Complement again to get A ∨ B
```

The other logic operations can be implemented by software in a similar fashion.

Shift Operations

The circular-shift operations are machine instructions in the basic computer. The other shifts of interest are the logical shifts and arithmetic shifts. These two shifts can be programmed with a small number of instructions.

The logical shift requires that zeros be added to the extreme positions. This is easily accomplished by clearing *E* and circulating the *AC* and *E*. Thus for a logical shift-right operation we need the two instructions

```
CLE
CIR
```

For a logical shift-left operation we need the two instructions

```
CLE
CIL
```

The arithmetic shifts depend on the type of representation of negative numbers. For the basic computer we have adopted the signed-2's complement representation. The rules for arithmetic shifts are listed in Sec. 4-6. For an arithmetic right-shift it is necessary that the sign bit in the leftmost position remain unchanged. But the sign bit itself is shifted into the high-order bit position of the number. The program for the arithmetic right-shift requires that we set *E* to the same value as the sign bit and circulate right, thus:

```
CLE  /Clear E to 0
SPA  /Skip if AC is positive; E remains 0
CME  /AC is negative; set E to 1
CIR  /Circulate E and AC
```

For arithmetic shift-left it is necessary that the added bit in the least significant position be 0. This is easily done by clearing *E* prior to the circulate-left operation. The sign bit must not change during this shift. With a circulate instruction, the sign bit moves into *E*. It is then necessary to compare the sign bit with the value of *E* after the operation. If the two values are equal, the arithmetic shift has been correctly implemented. If they are not equal, an overflow occurs. An overflow indicates that the unshifted number was too large. When multiplied by 2 (by means of the shift), the number so obtained exceeds the capacity of the *AC*.

6-7 Subroutines

Frequently, the same piece of code must be written over again in many different parts of a program. Instead of repeating the code every time it is needed, there is an obvious advantage if the common instructions are written only once. A set of common instructions that can be used in a program many times is called a *subroutine*. Each time that a subroutine is used in the main part of the program, a branch is executed to the beginning of the subroutine. After the subroutine has been executed, a branch is made back to the main program.

A subroutine consists of a self-contained sequence of instructions that carries out a given task. A branch can be made to the subroutine from any part of the main program. This poses the problem of how the subroutine knows which location to return to, since many different locations in the main program may make branches to the same subroutine. It is therefore necessary to store the return address somewhere in the computer for the subroutine to know where to return. Because branching to a subroutine and returning to the main program is such a common operation, all computers provide special instructions to facilitate subroutine entry and return.

In the basic computer, the link between the main program and a subroutine is the BSA instruction (branch and save return address). To explain how this instruction is used, let us write a subroutine that shifts the content of the accumulator four times to the left. Shifting a word four times is a useful operation for processing binary-coded decimal numbers or alphanumeric characters. Such an operation could have been included as a machine instruction in the computer. Since it is not included, a subroutine is formed to accomplish this task. The program of Table 6-16 starts by loading the value of X into the

TABLE 6-16 Program to Demonstrate the Use of Subroutines

Location			
		ORG 100	/Main program
100		LDA X	/Load X
101		BSA SH4	/Branch to subroutine
102		STA X	/Store shifted number
103		LDA Y	/Load Y
104		BSA SH4	/Branch to subroutine again
105		STA Y	/Store shifted number
106		HLT	
107	X,	HEX 1234	
108	Y,	HEX 4321	
			/Subroutine to shift left 4 times
109	SH4,	HEX 0	/Store return address here
10A		CIL	/Circulate left once
10B		CIL	
10C		CIL	
10D		CIL	/Circulate left fourth time
10E		AND MSK	/Set *AC*(13–16) to zero
10F		BUN SH4 I	/Return to main program
110	MSK,	HEX FFF0	/Mask operand
		END	

AC. The next instruction encountered is BSA SH4. The BSA instruction is in location 101. Subroutine SH4 must return to location 102 after it finishes its task. When the BSA instruction is executed, the control unit stores the return address 102 into the location defined by the symbolic address SH4 (which is 109). It also transfers the value of SH4 + 1 into the program counter. After this instruction is executed, memory location 109 contains the binary equivalent of hexadecimal 102 and the program counter contains the binary equivalent of hexadecimal 10A. This action has saved the return address and the subroutine is now executed starting from location 10A (since this is the content of *PC* in the next fetch cycle).

The computation in the subroutine circulates the content of *AC* four times to the left. In order to accomplish a logical shift operation, the four low-order bits must be set to zero. This is done by masking FFF0 with the content of *AC*. A mask operation is a logic AND operation that clears the bits of the *AC* where the mask operand is zero and leaves the bits of the *AC* unchanged where the mask operand bits are 1's.

The last instruction in the subroutine returns the computer to the main program. This is accomplished by the indirect branch instruction with an address symbol identical to the symbol used for the subroutine name. The address to which the computer branches is not SH4 but the value found in location SH4 because this is an indirect address instruction. What is found in location SH4 is the return address 102 which was previously stored there by the BSA instruction. The computer returns to execute the instruction in location 102. The main program continues by storing the shifted number into location X. A new number is then loaded into the *AC* from location Y, and another branch is made to the subroutine. This time location SH4 will contain the return address 105 since this is now the location of the next instruction after BSA. The new operand is shifted and the subroutine returns to the main program at location 105.

From this example we see that the first memory location of each subroutine serves as a link between the main program and the subroutine. The procedure for branching to a subroutine and returning to the main program is referred to as a subroutine *linkage*. The BSA instruction performs an operation commonly called subroutine *call*. The last instruction of the subroutine performs an operation commonly called subroutine *return*.

The procedure used in the basic computer for subroutine linkage is commonly found in computers with only one processor register. Many computers have multiple processor registers and some of them are assigned the name *index registers*. In such computers, an index register is usually employed to implement the subroutine linkage. A branch-to-subroutine instruction stores the return address in an index register. A return-from-subroutine instruction is effected by branching to the address presently stored in the index register.

Subroutine Parameters and Data Linkage

When a subroutine is called, the main program must transfer the data it wishes the subroutine to work with. In the previous example, the data were transferred through the accumulator. The operand was loaded into the *AC* prior to the branch. The subroutine shifted the number and left it there to be accepted by the main program. In general, it is necessary for the subroutine to have access to data from the calling program and to return results to that program. The accumulator can be used for a single input parameter and a single output parameter. In computers with multiple processor registers, more parameters can be transferred this way. Another way to transfer data to a subroutine is through the memory. Data are often placed in memory locations following the call. They can also be placed in a block of storage. The first address of the block is then placed in the memory location following the call. In any case, the return address always gives the link information for transferring data between the main program and the subroutine.

As an illustration, consider a subroutine that performs the logic OR operation. Two operands must be transferred to the subroutine and the subroutine must return the result of the operation. The accumulator can be used to transfer one operand and to receive the result. The other operand is inserted in the location following the BSA instruction. This is demonstrated in the program of Table 6-17. The first operand in location X is loaded into the *AC*. The second operand is stored in location 202 following the BSA instruction. After the branch, the first location in the subroutine holds the number 202. Note that in this case, 202 is not the return address but the address of the second operand. The subroutine starts performing the OR operation by complementing the first operand in the *AC* and storing it in a temporary location TMP. The second operand is loaded into the *AC* by an indirect instruction at location OR. Remember that location OR contains the number 202. When the instruction refers to it indirectly, the operand at location 202 is loaded into the *AC*. This operand is complemented and then ANDed with the operand stored in TMP. Complementing the result forms the OR operation.

The return from the subroutine must be manipulated so that the main program continues from location 203 where the next instruction is located. This is accomplished by incrementing location OR with the ISZ instruction. Now location OR holds the number 203 and an indirect BUN instruction causes a return to the proper place.

It is possible to have more than one operand following the BSA instruc-

TABLE 6-17 Program to Demonstrate Parameter Linkage

Location			
		ORG 200	
200		LDA X	/Load first operand into *AC*
201		BSA OR	/Branch to subroutine OR
202		HEX 3AF6	/Second operand stored here
203		STA Y	/Subroutine returns here
204		HLT	
205	X,	HEX 7B95	/First operand stored here
206	Y,	HEX 0	/Result stored here
207	OR,	HEX 0	/Subroutine OR
208		CMA	/Complement first operand
209		STA TMP	/Store in temporary location
20A		LDA OR I	/Load second operand
20B		CMA	/Complement second operand
20C		AND TMP	/AND complemented first operand
20D		CMA	/Complement again to get OR
20E		ISZ OR	/Increment return address
20F		BUN OR I	/Return to main program
210	TMP,	HEX 0	/Temporary storage
		END	

tion. The subroutine must increment the return address stored in its first location for each operand that it extracts from the calling program. Moreover, the calling program can reserve one or more locations for the subroutine to return results that are computed. The first location in the subroutine must be incremented for these locations as well, before the return. If there is a large amount of data to be transferred, the data can be placed in a block of storage and the address of the first item in the block is then used as the linking parameter.

A subroutine that moves a block of data starting at address 100 into a block starting with address 200 is listed in Table 6-18. The length of the block is 16 words. The first introduction is a branch to subroutine MVE. The first part of the subroutine transfers the three parameters 100, 200 and −16 from the main program and places them in its own storage location. The items are retrieved from their blocks by the use of two pointers. The counter ensures that only 16 items are moved. When the subroutine completes its operation, the data required is in the block starting from the location 200. The return to the main program is to the HLT instruction.

TABLE 6-18 Subroutine to Move a Block of Data

```
                             /Main program
          BSA MVE            /Branch to subroutine
          HEX 100            /First address of source data
          HEX 200            /First address of destination data
          DEC -16            /Number of items to move
          HLT
MVE,      HEX 0              /Subroutine MVE
          LDA MVE I          /Bring address of source
          STA PT1            /Store in first pointer
          ISZ MVE            /Increment return address
          LDA MVE I          /Bring address of destination
          STA PT2            /Store in second pointer
          ISZ MVE            /Increment return address
          LDA MVE I          /Bring number of items
          STA CTR            /Store in counter
          ISZ MVE            /Increment return address
LOP,      LDA PT1 I          /Load source item
          STA PT2 I          /Store in destination
          ISZ PT1            /Increment source pointer
          ISZ PT2            /Increment destination pointer
          ISZ CTR            /Increment counter
          BUN LOP            /Repeat 16 times
          BUN MVE I          /Return to main program
PT1,      —
PT2,      —
CTR,      —
```

6-8 Input–Output Programming

Users of the computer write programs with symbols that are defined by the programming language employed. The symbols are strings of characters and each character is assigned an 8-bit code so that it can be stored in computer memory. A binary-coded character enters the computer when an INP (input) instruction is executed. A binary-coded character is transferred to the output device when an OUT (output) instruction is executed. The output device detects the binary code and types the corresponding character.

Table 6-19(a) lists the instructions needed to input a character and store it in memory. The SKI instruction checks the input flag to see if a character is available for transfer. The next instruction is skipped if the input flag bit is 1. The INP instruction transfers the binary-coded character into *AC*(0–7). The character is then printed by means of the OUT instruction. A terminal unit that communicates directly with a computer does not print the character when a key is depressed. To type it, it is necessary to send an OUT instruction for the printer. In this way, the user is ensured that the correct transfer has occurred. If the SKI instruction finds the flag bit at 0, the next instruction in sequence is executed. This instruction is a branch to return and check the flag bit again. Because the input device is much slower than the computer, the two instructions in the loop will be executed many times before a character is transferred into the accumulator.

Table 6-19(b) lists the instructions needed to print a character initially stored in memory. The character is first loaded into the *AC*. The output flag is then checked. If it is 0, the computer remains in a two-instruction loop checking the flag bit. When the flag changes to 1, the character is transferred from the accumulator to the printer.

TABLE 6-19 Programs to Input and Output One Character

```
(a) Input a character:
   CIF,     SKI         /Check input flag
            BUN CIF     /Flag=0, branch to check again
            INP         /Flag=1, input character
            OUT         /Print character
            STA CHR     /Store character
            HLT
   CHR,     —           /Store character here
(b) Output one character:
            LDA CHR     /Load character into AC
   COF,     SKO         /Check output flag
            BUN COF     /Flag=0, branch to check again
            OUT         /Flag=1, output character
            HLT
   CHR,     HEX 0057    /Character is "W"
```

Character Manipulation

A computer is not just a calculator but also a symbol manipulator. The binary-coded characters that represent symbols can be manipulated by computer instructions to achieve various data-processing tasks. One such task may be to pack two characters in one word. This is convenient because each character occupies 8 bits and a memory word contains 16 bits. The program in Table 6-20 lists a subroutine named IN2 that inputs two characters and packs them into one 16-bit word. The packed word remains in the accumulator. Note that subroutine SH4 (Table 6-16) is called twice to shift the accumulator left eight times.

In the discussion of the assembler it was assumed that the symbolic program is stored in a section of memory which is sometimes called a *buffer*. The symbolic program being typed enters through the input device and is stored in consecutive memory locations in the buffer. The program listed in Table 6-21 can be used to input a symbolic program from the keyboard, pack two characters in one word, and store them in the buffer. The first address of the buffer is 500. The first double character is stored in location 500 and all characters are stored in sequential locations. The program uses a pointer for keeping track of the current empty location in the buffer. No counter is used in the program, so characters will be read as long as they are available or until the buffer reaches location 0 (after location FFFF). In a practical situation it may be necessary to limit the size of the buffer and a counter may be used for this purpose. Note that subroutine IN2 of Table 6-20 is called to input and pack the two characters.

In discussing the second pass of the assembler in Sec. 6-4 it was mentioned that one of the most common operations of an assembler is table lookup. This is an operation that searches a table to find out if it contains a given symbol. The search may be done by comparing the given symbol with each of the symbols stored in the table. The search terminates when a match occurs

TABLE 6-20 Subroutine to Input and Pack Two Characters

```
IN2,    —           /Subroutine entry
FST,    SKI
        BUN FST
        INP         /Input first character
        OUT
        BSA SH4     /Shift left four times
        BSA SH4     /Shift left four more times
SCD,    SKI
        BUN SCD
        INP         /Input second character
        OUT
        BUN IN2 I   /Return
```

TABLE 6-21 Program to Store Input Characters in a Buffer

```
        LDA ADS     /Load first address of buffer
        STA PTR     /Initialize pointer
LOP,    BSA IN2     /Go to subroutine IN2 (Table 6-20)
        STA PTR I   /Store double character word in buffer
        ISZ PTR     /Increment pointer
        BUN LOP     /Branch to input more characters
        HLT
ADS,    HEX 500     /First address of buffer
PTR,    HEX 0       /Location for pointer
```

or if none of the symbols match. When a match occurs, the assembler retrieves the equivalent binary value. A program for comparing two words is listed in Table 6-22. The comparison is accomplished by forming the 2's complement of a word (as if it were a number) and arithmetically adding it to the second word. If the result is zero, the two words are equal and a match occurs. If the result is not zero, the words are not the same. This program can serve as a subroutine in a table-lookup program.

Program Interrupt

The running time of input and output programs is made up primarily of the time spent by the computer in waiting for the external device to set its flag. The waiting loop that checks the flag keeps the computer occupied with a task that wastes a large amount of time. This waiting time can be eliminated if the interrupt facility is used to notify the computer when a flag is set. The advantage of using the interrupt is that the information transfer is initiated upon request from the external device. In the meantime, the computer can be busy performing other useful tasks. Obviously, if no other program resides in memory, there is nothing for the computer to do, so it might as well check for

TABLE 6-22 Program to Compare Two Words

```
        LDA WD1     /Load first word
        CMA
        INC         /Form 2's complement
        ADD WD2     /Add second word
        SZA         /Skip if AC is zero
        BUN UEQ     /Branch to "unequal" routine
        BUN EQL     /Branch to "equal" routine
WD1,    —
WD2,    —
```

the flags. The interrupt facility is useful in a multiprogram environment when two or more programs reside in memory at the same time.

Only one program can be executed at any given time even though two or more programs may reside in memory. The program currently being executed is referred to as the running program. The other programs are usually waiting for input or output data. The function of the interrupt facility is to take care of the data transfer of one (or more) program while another program is currently being executed. The running program must include an ION instruction to turn the interrupt on. If the interrupt facility is not used, the program must include an IOF instruction to turn it off. (The *start* switch of the computer should also turn the interrupt off.)

The interrupt facility allows the running program to proceed until the input or output device sets its ready flag. Whenever a flag is set to 1, the computer completes the execution of the instruction in progress and then acknowledges the interrupt. The result of this action is that the return address is stored in location 0. The instruction in location 1 is then performed; this initiates a service routine for the input or output transfer. The service routine can be stored anywhere in memory provided a branch to the start of the routine is stored in location 1. The service routine must have instructions to perform the following tasks:

1. Save contents of processor registers.
2. Check which flag is set.
3. Service the device whose flag is set.
4. Restore contents of processor registers.
5. Turn the interrupt facility on.
6. Return to the running program.

The contents of processor registers before the interrupt and after the return to the running program must be the same; otherwise, the running program may be in error. Since the service routine may use these registers, it is necessary to save their contents at the beginning of the routine and restore them at the end. The sequence by which the flags are checked dictates the priority assigned to each device. Even though two or more flags may be set at the same time, the devices nevertheless are serviced one at a time. The device with higher priority is serviced first followed by the one with lower priority.

The occurrence of an interrupt disables the facility from further interrupts. The service routine must turn the interrupt on before the return to the running program. This will enable further interrupts while the computer is executing the running program. The interrupt facility should not be turned on until after the return address is inserted into the program counter.

An example of a program that services an interrupt is listed in Table 6-23.

TABLE 6-23 Program to Service an Interrupt

Location			
0	ZRO,	—	/Return address stored here
1		BUN SRV	/Branch to service routine
100		CLA	/Portion of running program
101		ION	/Turn on interrupt facility
102		LDA X	
103		ADD Y	/Interrupt occurs here
104		STA Z	/Program returns here after interrupt
•		•	
•		•	
•			/Interrupt service routine
200	SRV,	STA SAC	/Store content of *AC*
		CIR	/Move *E* into *AC*(1)
		STA SE	/Store content of *E*
		SKI	/Check input flag
		BUN NXT	/Flag is off, check next flag
		INP	/Flag is on, input character
		OUT	/Print character
		STA PT1 I	/Store it in input buffer
		ISZ PT1	/Increment input pointer
	NXT,	SKO	/Check output flag
		BUN EXT	/Flag is off, exit
		LDA PT2 I	/Load character from output buffer
		OUT	/Output character
		ISZ PT2	/Increment output pointer
	EXT,	LDA SE	/Restore value of *AC*(1)
		CIL	/Shift it to *E*
		LDA SAC	/Restore content of *AC*
		ION	/Turn interrupt on
		BUN ZRO I	/Return to running program
	SAC,	—	/*AC* is stored here
	SE,	—	/*E* is stored here
	PT1,	—	/Pointer of input buffer
	PT2,	—	/Pointer of output buffer

Location 0 is reserved for the return address. Location 1 has a branch instruction to the beginning of the service routine SRV. The portion of the running program listed has an ION instruction that turns the interrupt on. Suppose that an interrupt occurs while the computer is executing the instruction in location 103. The interrupt cycle stores the binary equivalent of hexadecimal 104 in location 0 and branches to location 1. The branch instruction in location 1 sends the computer to the service routine SRV.

The service routine performs the six tasks mentioned above. The contents of *AC* and *E* are stored in special locations. (These are the only processor registers in the basic computer.) The flags are checked sequentially, the input flag first and the output flag second. If any or both flags are set, an item of data is transferred to or from the corresponding memory buffer. Before returning to the running program the previous contents of *E* and *AC* are restored and the interrupt facility is turned on. The last instruction causes a branch to the address stored in location 0. This is the return address stored there previously during the interrupt cycle. Hence the running program will continue from location 104, where it was interrupted.

A typical computer may have many more input and output devices connected to the interrupt facility. Furthermore, interrupt sources are not limited to input and output transfers. Interrupts can be used for other purposes, such as internal processing errors or special alarm conditions. Further discussion of interrupts and some advanced concepts concerning this important subject can be found in Sec. 11-5.

PROBLEMS

6-1. The following program is stored in the memory unit of the basic computer. Show the contents of the *AC*, *PC*, and *IR* (in hexadecimal), at the end, after each instruction is executed. All numbers listed below are in hexadecimal.

Location	Instruction
010	CLA
011	ADD 016
012	BUN 014
013	HLT
014	AND 017
015	BUN 013
016	C1A5
017	93C6

6-2. The following program is a list of instructions in hexadecimal code. The computer executes the instructions starting from address 100. What are the content of *AC* and the memory word at address 103 when the computer halts?

Location	Instruction
100	5103
101	7200
102	7001
103	0000
104	7800
105	7020
106	C103

6-3. List the assembly language program (of the equivalent binary instructions) generated by a compiler from the following Fortran program. Assume integer variables.

```
SUM = 0
SUM = SUM + A + B
DIF = DIF - C
SUM = SUM + DIF
```

6-4. Can the letter I be used as a symbolic address in the assembly language program defined for the basic computer? Justify the answer.

6-5. What happens during the first pass of the assembler (Fig. 6-1) if the line of code that has a pseudoinstruction ORG or END also has a label? Modify the flowchart to include an error message if this occurs.

6-6. A line of code in an assembly language program is as follows:

```
DEC -35
```

a. Show that four memory words are required to store the line of code and give their binary content.

b. Show that one memory word stores the binary translated code and give its binary content.

6-7. a. Obtain the address symbol table generated for the program of Table 6-13 during the first pass of the assembler.

b. List the translated program in hexadecimal.

6-8. The pseudoinstruction BSS N (block started by symbol) is sometimes employed to reserve *N* memory words for a group of operands. For example, the line of code

```
A,  BSS 10
```

informs the assembler that a block of 10 (decimal) locations is to be left free, starting from location A. This is similar to the Fortran statement DIMENSION A(10). Modify the flowchart of Fig. 6-1 to process this pseudoinstruction.

6-9. Modify the flowchart of Fig. 6-2 to include an error message when a symbolic address is not defined by a label.

6-10. Show how the MRI and non-MRI tables can be stored in memory.

6-11. List the assembly language program (of the equivalent binary instructions) generated by a compiler for the following IF statement:

```
IF(A - B) 10, 20, 30
```

The program branches to statement 10 if $A - B < 0$; to statement 20 if $A - B = 0$; and to statement 30 if $A - B > 0$.

6-12. **a.** Explain in words what the following program accomplishes when it is executed. What is the value of location CTR when the computer halts?
b. List the address symbol table obtained during the first pass of the assembler.
c. List the hexadecimal code of the translated program.

```
        ORG 100
        CLE
        CLA
        STA CTR
        LDA WRD
        SZA
        BUN ROT
        BUN STP
ROT,    CIL
        SZE
        BUN AGN
        BUN ROT
AGN,    CLE
        ISZ CTR
        SZA
        BUN ROT
STP,    HLT
CTR,    HEX 0
WRD,    HEX 62C1
        END
```

6-13. Write a program loop, using a pointer and a counter, that clears to 0 the contents of hexadecimal locations 500 through 5FF.

6-14. Write a program to multiply two positive numbers by a repeated addition method. For example, to multiply 5 × 4, the program evaluates the product by adding 5 four times, or 5 + 5 + 5 + 5.

6-15. The multiplication program of Table 6-14 is not initialized. After the program is executed once, location CTR will be left with zero. Show that if the program is executed again starting from location 100, the loop will be traversed 65536 times. Add the needed instructions to initialize the program.

6-16. Write a program to multiply two unsigned positive numbers, each with 16 significant bits, to produce an unsigned double-precision product.

6-17. Write a program to multiply two signed numbers with negative numbers being initially in signed-2's complement representation. The product should be single-precision and signed-2's complement representation if negative.

6-18. Write a program to subtract two double-precision numbers.

6-19. Write a program that evaluates the logic exclusive-OR of two logic operands.

6-20. Write a program for the arithmetic shift-left operation. Branch to OVF if an overflow occurs.

6-21. Write a subroutine to subtract two numbers. In the calling program, the BSA instruction is followed by the subtrahend and minuend. The difference is returned to the main program in the third location following the BSA instruction.

6-22. Write a subroutine to complement each word in a block of data. In the calling program, the BSA instruction is followed by two parameters: the starting address of the block and the number of words in the block.

6-23. Write a subroutine to circulate *E* and *AC* four times to the right. If *AC* contains hexadecimal 079C and $E = 1$, what are the contents of *AC* and *E* after the subroutine is executed?

6-24. Write a program to accept input characters, pack two characters in one word and store them in consecutive locations in a memory buffer. The first address of the buffer is $(400)_{16}$. The size of the buffer is $(512)_{10}$ words. If the buffer overflows, the computer should halt.

6-25. Write a program to unpack two characters from location WRD and store them in bits 0 through 7 of locations CH1 and CH2. Bits 9 through 15 should contain zeros.

6-26. Obtain a flowchart for a program to check for a CR code (hexadecimal 0D) in a memory buffer. The buffer contains two characters per word. When the code for CR is encountered, the program transfers it to bits 0 through 7 of location LNE without disturbing bits 8 through 15.

6-27. Translate the service routine SRV from Table 6-23 to its equivalent hexadecimal code. Assume that the routine is stored starting from location 200.

6-28. Write an interrupt service routine that performs all the required functions but the input device is serviced only if a special location, MOD, contains all 1's. The output device is serviced only if location MOD contains all 0's.

REFERENCES

1. Booth, T. L., *Introduction to Computer Engineering*, 3rd ed. New York: John Wiley, 1984.

2. Gear, C. W., *Computer Organization and Programming*, 3rd ed. New York: McGraw-Hill, 1980.

3. Gibson, G. A., *Computer Systems Concepts and Design*. Englewood Cliffs, NJ: Prentice Hall, 1991.
4. Gray, N. A. B., *Introduction to Computer Systems*. Englewood Cliffs, NJ: Prentice Hall, 1987.
5. Levy, H. M., and R. H. Eckhouse, Jr., *Computer Programming and Architecture: The VAX-11*. Bedford, MA: Digital Press, 1980.
6. Lewin, M. H., *Logic Design and Computer Organization*. Reading, MA: Addison-Wesley, 1983.
7. Prosser, F. P., and D. E. Winkel, *The Art of Digital Design*, 2nd ed. Englewood Cliffs, NJ: Prentice Hall, 1987.
8. Shiva, S. G., *Computer Design and Architecture*, 2nd ed. New York: HarperCollins Publishers, 1991.
9. Tanenbaum, A. S., *Structured Computer Organization*, 3rd ed. Englewood Cliffs, NJ: Prentice Hall, 1990.
10. Wakerly, J. F., *Microcomputer Architecture and Programming*. New York: John Wiley, 1981.