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 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.

hexadecimal code

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.