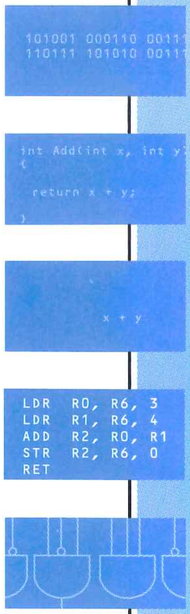
chapter

7



Assembly Language

By now, you are probably a little tired of 1s and 0s and keeping track of 0001 meaning ADD and 1001 meaning NOT. Also, wouldn't it be nice if we could refer to a memory location by some meaningful symbolic name instead of memorizing its 16-bit address? And wouldn't it be nice if we could represent each instruction in some more easily comprehensible way, instead of having to keep track of which bit of the instruction conveys which individual piece of information about the instruction. It turns out that help is on the way.

In this chapter, we introduce assembly language, a mechanism that does all that, and more.

7.1 Assembly Language Programming—Moving Up a Level

Recall the levels of transformation identified in Figure 1.6 of Chapter 1. Algorithms are transformed into programs described in some mechanical language. This mechanical language can be, as it is in Chapter 5, the machine language of a particular computer. Recall that a program is in a computer's machine language if every instruction in the program is from the ISA of that computer.

On the other hand, the mechanical language can be more user-friendly. We generally partition mechanical languages into two classes, high-level and low-level. Of the two, high-level languages are much more user-friendly. Examples are C, C++, Java, Fortran, COBOL, Pascal, plus more than a thousand others. Instructions in a high-level language almost (but not quite) resemble statements in a natural language such as English. High-level languages tend to be ISA independent. That is, once you learn how to program in C (or Fortran or Pascal) for one ISA, it is a small step to write programs in C (or Fortran or Pascal) for another ISA.

Before a program written in a high-level language can be executed, it must be translated into a program in the ISA of the computer on which it is expected to execute. It is usually the case that each statement in the high-level language specifies several instructions in the ISA of the computer. In Chapter 11, we will introduce the high-level language C, and in Chapters 12 through 19, we will show the relationship between various statements in C and their corresponding translations in LC-3 code. In this chapter, however, we will only move up a small notch from the ISA we dealt with in Chapter 5.

A small step up from the ISA of a machine is that ISA's assembly language. Assembly language is a low-level language. There is no confusing an instruction in a low-level language with a statement in English. Each assembly language instruction usually specifies a single instruction in the ISA. Unlike high-level languages, which are usually ISA independent, low-level languages are very much ISA dependent. In fact, it is usually the case that each ISA has only one assembly language.

The purpose of assembly language is to make the programming process more user-friendly than programming in machine language (i.e., the ISA of the computer with which we are dealing), while still providing the programmer with detailed control over the instructions that the computer can execute. So, for example, while still retaining control over the detailed instructions the computer is to carry out, we are freed from having to remember what opcode is 0001 and what opcode is 1001, or what is being stored in memory location 0011111100001010 and what is being stored in location 0011111100000101. Assembly languages let us use mnemonic devices for opcodes, such as ADD and NOT, and they let us give meaningful symbolic names to memory locations, such as SUM or PRODUCT, rather than use their 16-bit addresses. This makes it easier to differentiate which memory location is keeping track of a SUM and which memory location is keeping track of a PRODUCT. We call these names *symbolic addresses*.

We will see, starting in Chapter 11, that when we take the larger step of moving up to a higher-level language (such as C), programming will be even more user-friendly, but we will relinquish control of exactly which detailed instructions are to be carried out in behalf of a high-level language statement.

7.2 An Assembly Language Program

We will begin our study of the LC-3 assembly language by means of an example. The program in Figure 7.1 multiplies the integer intially stored in NUMBER by 6 by adding the integer to itself six times. For example, if the integer is 123, the program computes the product by adding 123 + 123 + 123 + 123 + 123 + 123.

The program consists of 21 lines of code. We have added a *line number* to each line of the program in order to be able to refer to individual lines easily. This is a common practice. These line numbers are not part of the program. Ten lines start with a semicolon, designating that they are strictly for the benefit of the human reader. More on this momentarily. Seven lines (06, 07, 08, 0C, 0D,

```
01  ;
02  ; Program to multiply an integer by the constant 6.
03  ; Before execution, an integer must be stored in NUMBER.
04  ;
05          .ORIG   x3050
06          LD      R1,SIX
07          LD      R2,NUMBER
08          AND     R3,R3,#0      ; Clear R3. It will
09                                ; contain the product.
0A  ; The inner loop
0B  ;
0C  AGAIN   ADD     R3,R3,R2
0D          ADD     R1,R1,#-1     ; R1 keeps track of
0E          BRp     AGAIN         ; the iterations
0F  ;
10          HALT
11  ;
12  NUMBER  .BLKW   1
13  SIX     .FILL   x0006
14  ;
15          .END
```

Figure 7.1 An assembly language program

0E, and 10) specify assembly language instructions to be translated into machine language instructions of the LC-3, which will actually be carried out when the program runs. The remaining four lines (05, 12, 13, and 15) contain pseudo-ops, which are messages from the programmer to the translation program to help in the translation process. The translation program is called an *assembler* (in this case the LC-3 assembler), and the translation process is called *assembly*.

7.2.1 Instructions

Instead of an instruction being 16 0s and 1s, as is the case in the LC-3 ISA, an instruction in assembly language consists of four parts, as follows:

```
LABEL   OPCODE   OPERANDS   ; COMMENTS
```

Two of the parts (LABEL and COMMENTS) are optional. More on that momentarily.

Opcodes and Operands

Two of the parts (OPCODE and OPERANDS) are **mandatory**. An instruction must have an OPCODE (the thing the instruction is to do), and the appropriate number of OPERANDS (the things it is supposed to do it to). Not surprisingly, this was exactly what we encountered in Chapter 5 when we studied the LC-3 ISA.

The OPCODE is a symbolic name for the opcode of the corresponding LC-3 instruction. The idea is that it is easier to remember an operation by the symbolic name ADD, AND, or LDR than by the 4-bit quantity 0001, 0101, or 0110. Figure 5.3 (also Figure A.2) lists the OPCODES of the 15 LC-3 instructions. Pages 526 through 541 show the assembly language representations for the 15 LC-3 instructions.

The number of operands depends on the operation being performed. For example, the ADD instruction (line 0C) requires three operands (two sources to obtain the numbers to be added, and one destination to designate where the result is to be placed). All three operands must be explicitly identified in the instruction.

```
AGAIN    ADD    R3,R3,R2
```

The operands to be added are obtained from register 2 and from register 3. The result is to be placed in register 3. We represent each of the registers 0 through 7 as R0, R1, R2, . . . , R7.

The LD instruction (line 07) requires two operands (the memory location from which the value is to be read and the destination register that is to contain the value after the instruction completes its execution). We will see momentarily that memory locations will be given symbolic addresses called *labels*. In this case, the location from which the value is to be read is given the label *NUMBER*. The destination into which the value is to be loaded is register 2.

```
LD   R2, NUMBER
```

As we discussed in Section 5.1.6, operands can be obtained from registers, from memory, or they may be literal (i.e., immediate) values in the instruction. In the case of register operands, the registers are explicitly represented (such as R2 and R3 in line 0C). In the case of memory operands, the symbolic name of the memory location is explicitly represented (such as NUMBER in line 07 and SIX in line 06). In the case of immediate operands, the actual value is explicitly represented (such as the value 0 in line 08).

```
AND  R3, R3, #0 ; Clear R3. It will contain the product.
```

A literal value must contain a symbol identifying the representation base of the number. We use # for decimal, x for hexadecimal, and b for binary. Sometimes there is no ambiguity, such as in the case 3F0A, which is a hex number. Nonetheless, we write it as x3F0A. Sometimes there is ambiguity, such as in the case 1000. x1000 represents the decimal number 4096, b1000 represents the decimal number 8, and #1000 represents the decimal number 1000.

Labels

Labels are symbolic names that are used to identify memory locations that are referred to explicitly in the program. In LC-3 assembly language, a label consists of from one to 20 alphanumeric characters (i.e., a capital or lowercase letter of the alphabet, or a decimal digit), starting with a letter of the alphabet. NOW, Under21, R2D2, and C3PO are all examples of possible LC-3 assembly language labels.

There are two reasons for explicitly referring to a memory location.

1. The location contains the target of a branch instruction (for example, AGAIN in line 0C).

2. The location contains a value that is loaded or stored (for example, NUMBER, line 12, and SIX, line 13).

The location AGAIN is specifically referenced by the branch instruction in line 0E.

```
BRp    AGAIN
```

If the result of ADD R1,R1,#–1 is positive (as evidenced by the P condition code being set), then the program branches to the location explicitly referenced as AGAIN to perform another iteration.

The location NUMBER is specifically referenced by the load instruction in line 07. The value stored in the memory location explicitly referenced as NUMBER is loaded into R2.

If a location in the program is not explicitly referenced, then there is no need to give it a label.

Comments

Comments are messages intended only for human consumption. They have no effect on the translation process and indeed are not acted on by the LC-3 assembler. They are identified in the program by semicolons. A semicolon signifies that the rest of the line is a comment and is to be ignored by the assembler. If the semicolon is the first nonblank character on the line, the entire line is ignored. If the semicolon follows the operands of an instruction, then only the comment is ignored by the assembler.

The purpose of comments is to make the program more comprehensible to the human reader. They help explain a nonintuitive aspect of an instruction or a set of instructions. In lines 08 and 09, the comment "Clear R3; it will contain the product" lets the reader know that the instruction on line 08 is initializing R3 prior to accumulating the product of the two numbers. While the purpose of line 08 may be obvious to the programmer today, it may not be the case two years from now, after the programmer has written an additional 30,000 lines of code and cannot remember why he/she wrote AND R3,R3,#0. It may also be the case that two years from now, the programmer no longer works for the company and the company needs to modify the program in response to a product update. If the task is assigned to someone who has never seen the code before, comments go a long way toward improving comprehension.

It is important to make comments that provide additional insight and not just restate the obvious. There are two reasons for this. First, comments that restate the obvious are a waste of everyone's time. Second, they tend to obscure the comments that say something important because they add clutter to the program. For example, in line 0D, the comment "Decrement R1" would be a bad idea. It would provide no additional insight to the instruction, and it would add clutter to the page.

Another purpose of comments, and also the judicious use of extra blank spaces to a line, is to make the visual presentation of a program easier to understand. So, for example, comments are used to separate pieces of the program from each other to make the program more readable. That is, lines of code that work together to compute a single result are placed on successive lines, while pieces of a program that produce separate results are separated from each other. For example, note that lines 0C through 0E are separated from the rest of the code by lines 0B and 0F. There is nothing on lines 0B and 0F other than the semicolons.

Extra spaces that are ignored by the assembler provide an opportunity to align elements of a program for easier readability. For example, all the opcodes start in the same column on the page.

7.2.2 Pseudo-ops (Assembler Directives)

The LC-3 assembler is a program that takes as input a string of characters representing a computer program written in LC-3 assembly language and translates it into a program in the ISA of the LC-3. Pseudo-ops are helpful to the assembler in performing that task.

Actually, a more formal name for a pseudo-op is *assembler directive*. They are called pseudo-ops because they do not refer to operations that will be performed by the program during execution. Rather, the pseudo-op is strictly a message to the assembler to help the assembler in the assembly process. Once the assembler handles the message, the pseudo-op is discarded. The LC-3 assembler contains five pseudo-ops: .ORIG, .FILL, .BLKW, .STRINGZ, and .END. All are easily recognizable by the dot as their first character.

.ORIG

.ORIG tells the assembler where in memory to place the LC-3 program. In line 05, .ORIG x3050 says, start with location x3050. As a result, the LD R1,SIX instruction will be put in location x3050.

.FILL

.FILL tells the assembler to set aside the next location in the program and initialize it with the value of the operand. In line 13, the ninth location in the resultant LC-3 program is initialized to the value x0006.

.BLKW

.BLKW tells the assembler to set aside some number of sequential memory locations (i.e., a **BL**oc**K** of **W**ords) in the program. The actual number is the operand of the .BLKW pseudo-op. In line 12, the pseudo-op instructs the assembler to set aside one location in memory (and also to label it NUMBER, incidentally).

The pseudo-op .BLKW is particularly useful when the actual value of the operand is not yet known. For example, one might want to set aside a location in memory for storing a character input from a keyboard. It will not be until the program is run that we will know the identity of that keystroke.

.STRINGZ

.STRINGZ tells the assembler to initialize a sequence of $n+1$ memory locations. The argument is a sequence of n characters, inside double quotation marks. The first *n* words of memory are initialized with the zero-extended ASCII codes of the corresponding characters in the string. The final word of memory is initialized to 0. The last character, x0000, provides a convenient sentinel for processing the string of ASCII codes.

For example, the code fragment

```
        .ORIG     x3010
HELLO   .STRINGZ  "Hello, World!"
```

would result in the assembler initializing locations x3010 through x301D to the following values:

```
x3010: x0048
x3011: x0065
x3012: x006C
x3013: x006C
x3014: x006F
x3015: x002C
x3016: x0020
x3017: x0057
x3018: x006F
x3019: x0072
x301A: x006C
x301B: x0064
x301C: x0021
x301D: x0000
```

.END

.END tells the assembler where the program ends. Any characters that come after .END will not be used by the assembler. *Note:* .END does not stop the program during execution. In fact, .END does not even exist at the time of execution. It is simply a delimiter—it marks the end of the source program.

7.2.3 Example: The Character Count Example of Section 5.5, Revisited

Now we are ready for a complete example. Let's consider again the problem of Section 5.5. We wish to write a program that will take a character that is input from the keyboard and a file and count the number of occurrences of that character in that file. As before, we first develop the algorithm by constructing the flowchart. Recall that in Section 6.1, we showed how to decompose the problem systematically so as to generate the flowchart of Figure 5.16. In fact, the final step of that process in Chapter 6 is the flowchart of Figure 6.3e, which is essentially identical to Figure 5.16. Next, we use the flowchart to write the actual program. This time, however, we enjoy the luxury of not worrying about 0s and 1s and instead write the program in LC-3 assembly language. The program is shown in Figure 7.2.

```
01   ;
02   ; Program to count occurrences of a character in a file.
03   ; Character to be input from the keyboard.
04   ; Result to be displayed on the monitor.
05   ; Program works only if no more than 9 occurrences are found.
06   ;
07   ;
08   ; Initialization
09   ;
0A           .ORIG    x3000
0B           AND      R2,R2,#0      ; R2 is counter, initialize to 0
0C           LD       R3,PTR        ; R3 is pointer to characters
0D           TRAP     x23           ; R0 gets character input
0E           LDR      R1,R3,#0      ; R1 gets the next character
0F   ;
10   ; Test character for end of file
11   ;
13   TEST    ADD      R4,R1,#-4     ; Test for EOT
14           BRz      OUTPUT        ; If done, prepare the output
15   ;
16   ; Test character for match.  If a match, increment count.
17   ;
18           NOT      R1,R1
19           ADD      R1,R1,R0      ; If match, R1 = xFFFF
1A           NOT      R1,R1         ; If match, R1 = x0000
1B           BRnp     GETCHAR       ; If no match, do not increment
1C           ADD      R2,R2,#1
1D   ;
1E   ; Get next character from the file
1F   ;
20   GETCHAR ADD      R3,R3,#1      ; Increment the pointer
21           LDR      R1,R3,#0      ; R1 gets the next character to test
22           BRnzp    TEST
23   ;
24   ; Output the count.
25   ;
26   OUTPUT  LD       R0,ASCII      ; Load the ASCII template
27           ADD      R0,R0,R2      ; Convert binary to ASCII
28           TRAP     x21           ; ASCII code in R0 is displayed
29           TRAP     x25           ; Halt machine
2A   ;
2B   ; Storage for pointer and ASCII template
2C   ;
2D   ASCII   .FILL    x0030
2E   PTR     .FILL    x4000
2F           .END
```

Figure 7.2 The assembly language program to count occurrences of a character

A few notes regarding this program:

Three times during this program, assistance in the form of a service call is required of the operating system. In each case, a TRAP instruction is used. TRAP x23 causes a character to be input from the keyboard and placed in R0 (line 0D). TRAP x21 causes the ASCII code in R0 to be displayed on the monitor (line 28). TRAP x25 causes the machine to be halted (line 29). As we said before, we will leave the details of how the TRAP instruction is carried out until Chapter 9.

The ASCII codes for the decimal digits 0 to 9 (0000 to 1001) are x30 to x39. The conversion from binary to ASCII is done simply by adding x30 to the binary value of the decimal digit. Line 2D shows the label ASCII used to identify the memory location containing x0030.

The file that is to be examined starts at address x4000 (see line 2E). Usually, this starting address would not be known to the programmer who is writing this program since we would want the program to work on files that will become available in the future. That situation will be discussed in Section 7.4.

7.3 The Assembly Process

7.3.1 Introduction

Before an LC-3 assembly language program can be executed, it must first be translated into a machine language program, that is, one in which each instruction is in the LC-3 ISA. It is the job of the LC-3 assembler to perform that translation.

If you have available an LC-3 assembler, you can cause it to translate your assembly language program into a machine language program by executing an appropriate command. In the LC-3 assembler that is generally available via the Web, that command is *assemble* and requires as an argument the filename of your assembly language program. For example, if the filename is solution1.asm, then

```
assemble solution1.asm outfile
```

produces the file outfile, which is in the ISA of the LC-3. It is necessary to check with your instructor for the correct command line to cause the LC-3 assembler to produce a file of 0s and 1s in the ISA of the LC-3.

7.3.2 A Two-Pass Process

In this section, we will see how the assembler goes through the process of translating an assembly language program into a machine language program. We will use as our input to the process the assembly language program of Figure 7.2.

You remember that there is in general a one-to-one correspondence between instructions in an assembly language program and instructions in the final machine language program. We could try to perform this translation in one pass through the assembly language program. Starting from the top of Figure 7.2, the assembler discards lines 01 to 09, since they contain only comments. Comments are strictly for human consumption; they have no bearing on the translation process. The assembler then moves on to line 0A. Line 0A is a pseudo-op; it tells the assembler that the machine language program is to start at location x3000. The assembler then moves on to line 0B, which it can easily translate into LC-3 machine code. At this point, we have

```
x3000:   0101010010100000
```

The LC-3 assembler moves on to translate the next instruction (line 0C). Unfortunately, it is unable to do so since it does not know the meaning of the symbolic address PTR. At this point the assembler is stuck, and the assembly process fails.

To prevent this from occurring, the assembly process is done in two complete passes (from beginning to .END) through the entire assembly language program. The objective of the first pass is to identify the actual binary addresses corresponding to the symbolic names (or labels). This set of correspondences is known as the *symbol table*. In pass 1, we construct the symbol table. In pass 2, we translate the individual assembly language instructions into their corresponding machine language instructions.

Thus, when the assembler examines line 0C for the purpose of translating

```
LD R3,PTR
```

during the second pass, it already knows the correspondence between PTR and x3013 (from the first pass). Thus it can easily translate line 0C to

```
x3001:   0010011000010001
```

The problem of not knowing the 16-bit address corresponding to PTR no longer exists.

7.3.3 The First Pass: Creating the Symbol Table

For our purposes, the symbol table is simply a correspondence of symbolic names with their 16-bit memory addresses. We obtain these correspondences by passing through the assembly language program once, noting which instruction is assigned to which address, and identifying each label with the address of its assigned entry.

Recall that we provide labels in those cases where we have to refer to a location, either because it is the target of a branch instruction or because it contains data that must be loaded or stored. Consequently, if we have not made any programming mistakes, and if we identify all the labels, we will have identified all the symbolic addresses used in the program.

The preceding paragraph assumes that our entire program exists between our .ORIG and .END pseudo-ops. This is true for the assembly language program of Figure 7.2. In Section 7.4, we will consider programs that consist of multiple parts, each with its own .ORIG and .END, wherein each part is assembled separately.

The first pass starts, after discarding the comments on lines 01 to 09, by noting (line 0A) that the first instruction will be assigned to address x3000. We keep track of the location assigned to each instruction by means of a location counter (LC). The LC is initialized to the address specified in .ORIG, that is, x3000.

The assembler examines each instruction in sequence and increments the LC once for each assembly language instruction. If the instruction examined contains a label, a symbol table entry is made for that label, specifying the current contents of LC as its address. The first pass terminates when the .END instruction is encountered.

The first instruction that has a label is at line 13. Since it is the fifth instruction in the program and since the LC at that point contains x3004, a symbol table entry is constructed thus:

Symbol	Address
TEST	x3004

The second instruction that has a label is at line 20. At this point, the LC has been incremented to x300B. Thus a symbol table entry is constructed, as follows:

Symbol	Address
GETCHAR	x300B

At the conclusion of the first pass, the symbol table has the following entries:

Symbol	Address
TEST	x3004
GETCHAR	x300B
OUTPUT	x300E
ASCII	x3012
PTR	x3013

7.3.4 The Second Pass: Generating the Machine Language Program

The second pass consists of going through the assembly language program a second time, line by line, this time with the help of the symbol table. At each line, the assembly language instruction is translated into an LC-3 machine language instruction.

Starting again at the top, the assembler again discards lines 01 through 09 because they contain only comments. Line 0A is the .ORIG pseudo-op, which the assembler uses to initialize LC to x3000. The assembler moves on to line 0B and produces the machine language instruction 0101010010100000. Then the assembler moves on to line 0C.

This time, when the assembler gets to line 0C, it can completely assemble the instruction since it knows that PTR corresponds to x3013. The instruction is LD, which has an opcode encoding of 0010. The destination register (DR) is R3, that is, 011.

PCoffset is computed as follows: We know that PTR is the label for address x3013, and that the incremented PC is LC+1, in this case x3002. Since PTR (x3013) must be the sum of the incremented PC (x3002) and the sign-extended PCoffset, PCoffset must be x0011. Putting this all together, x3001 is set to 0010011000010001, and the LC is incremented to x3002.

Note: In order to use the LD instruction, it is necessary that the source of the load, in this case the address whose label is PTR, is not more than +256 or −255 memory locations from the LD instruction itself. If the address of PTR had been greater than LC+1 +255 or less than LC+1 −256, then the offset would not fit in bits [8:0] of the instruction. In such a case, an assembly error would have occurred, preventing the assembly process from finishing successfully. Fortunately, PTR is close enough to the LD instruction, so the instruction assembled correctly.

The second pass continues. At each step, the LC is incremented and the location specified by LC is assigned the translated LC-3 instruction or, in the case of .FILL, the value specified. When the second pass encounters the .END instruction, assembly terminates.

The resulting translated program is shown in Figure 7.3.

Address	Binary
	0011000000000000
x3000	0101010010100000
x3001	0010011000010001
x3002	1111000000100011
x3003	0110001011000000
x3004	0001100001111100
x3005	0000010000001000
x3006	1001001001111111
x3007	0001001001000000
x3008	1001001001111111
x3009	0000101000000001
x300A	0001010010100001
x300B	0001011011100001
x300C	0110001011000000
x300D	0000111111110110
x300E	0010000000000011
x300F	0001000000000010
x3010	1111000000100001
x3011	1111000000100101
x3012	0000000000110000
x3013	0100000000000000

Figure 7.3 The machine language program for the assembly language program of Figure 7.2

That process was, on a good day, merely tedious. Fortunately, you do not have to do it for a living—the LC-3 assembler does that. And, since you now know LC-3 assembly language, there is no need to program in machine language. Now we can write our programs symbolically in LC-3 assembly language and invoke the LC-3 assembler to create the machine language versions that can execute on an LC-3 computer.

7.4 Beyond the Assembly of a Single Assembly Language Program

Our purpose in this chapter has been to take you up one more notch from the ISA of the computer and introduce assembly language. Although it is still quite a large step from C or C++, assembly language does, in fact, save us a good deal of pain. We have also shown how a rudimentary two-pass assembler actually works to translate an assembly language program into the machine language of the LC-3 ISA.

There are many more aspects to sophisticated assembly language programming that go well beyond an introductory course. However, our reason for teaching assembly language is not to deal with its sophistication, but rather to show its innate simplicity. Before we leave this chapter, however, there are a few additional highlights we should explore.

7.4.1 The Executable Image

When a computer begins execution of a program, the entity being executed is called an *executable image*. The executable image is created from modules often created independently by several different programmers. Each module is translated separately into an object file. We have just gone through the process of performing that translation ourselves by mimicking the LC-3 assembler. Other modules, some written in C perhaps, are translated by the C compiler. Some modules are written by users, and some modules are supplied as library routines by the operating system. Each object file consists of instructions in the ISA of the computer being used, along with its associated data. The final step is to *link* all the object modules together into one executable image. During execution of the program, the FETCH, DECODE, ... instruction cycle is applied to instructions in the executable image.

7.4.2 More than One Object File

It is very common to form an executable image from more than one object file. In fact, in the real world, where most programs invoke libraries provided by the operating system as well as modules generated by other programmers, it is much more common to have multiple object files than a single one.

A case in point is our example character count program. The program counts the number of occurrences of a character in a file. A typical application could easily have the program as one module and the input data file as another. If this were the case, then the starting address of the file, shown as x4000 in line 2E of Figure 7.2, would not be known when the program was written. If we replace line 2E with

```
PTR   .FILL   STARTofFILE
```

then the program of Figure 7.2 will not assemble because there will be no symbol table entry for STARTofFILE. What can we do?

If the LC-3 assembly language, on the other hand, contained the pseudo-op .EXTERNAL, we could identify STARTofFILE as the symbolic name of an address that is not known at the time the program of Figure 7.2 is assembled. This would be done by the following line

```
.EXTERNAL    STARTofFILE,
```

which would send a message to the LC-3 assembler that the absence of label STARTofFILE is not an error in the program. Rather, STARTofFILE is a label in some other module that will be translated independently. In fact, in our case, it will be the label of the location of the first character in the file to be examined by our character count program.

If the LC-3 assembly language had the pseudo-op .EXTERNAL, and if we had designated STARTofFILE as .EXTERNAL, the LC-3 would be able to create a symbol table entry for STARTofFILE, and instead of assigning it an address, it would mark the symbol as belonging to another module. At *link time,* when all the modules are combined, the linker (the program that manages the "combining" process) would use the symbol table entry for STARTofFILE in another module to complete the translation of our revised line 2E.

In this way, the .EXTERNAL pseudo-op allows references by one module to symbolic locations in another module without a problem. The proper translations are resolved by the linker.

Exercises

7.1 An assembly language program contains the following two instructions. The assembler puts the translated version of the LDI instruction that follows into location x3025 of the object module. After assembly is complete, what is in location x3025?

```
PLACE    .FILL    x45A7
         LDI      R3, PLACE
```

7.2 An LC-3 assembly language program contains the instruction:

```
ASCII   LD  R1, ASCII
```

The symbol table entry for ASCII is x4F08. If this instruction is executed during the running of the program, what will be contained in R1 immediately after the instruction is executed?

7.3 What is the problem with using the string AND as a label?

7.4 Create the symbol table entries generated by the assembler when translating the following routine into machine code:

```
         .ORIG    x301C
         ST       R3, SAVE3
         ST       R2, SAVE2
         AND      R2, R2, #0
TEST     IN
         BRz      TEST
         ADD      R1, R0, #-10
         BRn      FINISH
         ADD      R1, R0, #-15
         NOT      R1, R1
         BRn      FINISH
         HALT
FINISH   ADD      R2, R2, #1
         HALT
SAVE3    .FILL    X0000
SAVE2    .FILL    X0000
         .END
```

7.5 *a.* What does the following program do?

```
        .ORIG   x3000
        LD      R2, ZERO
        LD      R0, M0
        LD      R1, M1
LOOP    BRz     DONE
        ADD     R2, R2, R0
        ADD     R1, R1, -1
        BR      LOOP
DONE    ST      R2, RESULT
        HALT
RESULT  .FILL   x0000
ZERO    .FILL   x0000
M0      .FILL   x0004
M1      .FILL   x0803
        .END
```

b. What value will be contained in RESULT after the program runs to completion?

7.6 Our assembler has crashed and we need your help! Create a symbol table and assemble the instructions at labels D, E, and F for the program below. You may assume another module deposits a positive value into A before this module executes.

```
        .ORIG   x3000
        AND     R0, R0, #0
D       LD      R1, A
        AND     R2, R1, #1
        BRp     B
E       ADD     R1, R1, #-1
B       ADD     R0, R0, R1
        ADD     R1, R1, #-2
F       BRp     B
        ST      R0, C
        TRAP    x25
A       .BLKW   1
C       .BLKW   1
        .END
```

In no more than 15 words, what does the above program do?

7.7 Write an LC-3 assembly language program that counts the number of 1s in the value stored in R0 and stores the result into R1. For example, if R0 contains 0001001101110000, then after the program executes, the result stored in R1 would be 0000 0000 0000 0110.

7.8 An engineer is in the process of debugging a program she has written. She is looking at the following segment of the program, and decides to place a breakpoint in memory at location 0xA404. Starting with the PC = 0xA400, she initializes all the registers to zero and runs the program until the breakpoint is encountered.

```
Code Segment:
...
0xA400  THIS1  LEA    R0, THIS1
0xA401  THIS2  LD     R1, THIS2
0xA402  THIS3  LDI    R2, THIS5
0xA403  THIS4  LDR    R3, R0, #2
0xA404  THIS5  .FILL  xA400
...
```

Show the contents of the register file (in hexadecimal) when the breakpoint is encountered.

7.9 What is the purpose of the .END pseudo-op? How does it differ from the HALT instruction?

7.10 The following program fragment has an error in it. Identify the error and explain how to fix it.

```
            ADD      R3, R3, #30
            ST       R3, A
            HALT
A           .FILL    #0
```

Will this error be detected when this code is assembled or when this code is run on the LC-3?

7.11 The LC-3 assembler must be able to convert constants represented in ASCII into their appropriate binary values. For instance, x2A translates into 00101010 and #12 translates into 00001100. Write an LC-3 assembly language program that reads a decimal or hexadecimal constant from the keyboard (i.e., it is preceded by a # character signifying it is a decimal, or x signifying it is hex) and prints out the binary representation. Assume the constants can be expressed with no more than two decimal or hex digits.

7.12 What does the following LC-3 program do?

```
            .ORIG   x3000
            AND     R5, R5, #0
            AND     R3, R3, #0
            ADD     R3, R3, #8
            LDI     R1, A
            ADD     R2, R1, #0
AG          ADD     R2, R2, R2
            ADD     R3, R3, #-1
            BRnp    AG
            LD      R4, B
            AND     R1, R1, R4
            NOT     R1, R1
            ADD     R1, R1, #1
            ADD     R2, R2, R1
            BRnp    NO
            ADD     R5, R5, #1
NO          HALT
B           .FILL   xFF00
A           .FILL   x4000
            .END
```

7.13 The following program adds the values stored in memory locations A, B, and C, and stores the result into memory. There are two errors in the code. For each, describe the error and indicate whether it will be detected at assembly time or at run time.

```
Line No.
1                     .ORIG  x3000
2          ONE        LD  R0, A
3                     ADD R1, R1, R0
4          TWO        LD  R0, B
5                     ADD R1, R1, R0
6          THREE      LD  R0, C
7                     ADD R1, R1, R0
8                     ST  R1, SUM
9                     TRAP  x25
10         A          .FILL x0001
11         B          .FILL x0002
12         C          .FILL x0003
13         D          .FILL x0004
14                    .END
```

7.14 *a.* Assemble the following program:

```
              .ORIG    x3000
              STI      R0, LABEL
              OUT
              HALT
    LABEL     .STRINGZ "%"
              .END
```

b. The programmer intended the program to output a % to the monitor, and then halt. Unfortunately, the programmer got confused about the semantics of each of the opcodes (that is, exactly what function is carried out by the LC-3 in response to each opcode). Replace exactly **one** opcode in this program with the correct opcode to make the program work as intended.

c. The original program from part *a* was executed. However, execution exhibited some very strange behavior. The strange behavior was in part due to the programming error, and in part due to the fact that the value in R0 when the program started executing was x3000. Explain what the strange behavior was and why the program behaved that way.

7.15 The following is an LC-3 program that performs a function. Assume a sequence of integers is stored in consecutive memory locations, one integer per memory location, starting at the location x4000. The sequence terminates with the value x0000. What does the following program do?

```
           .ORIG    x3000
           LD       R0, NUMBERS
           LD       R2, MASK
LOOP       LDR      R1, R0, #0
           BRz      DONE
           AND      R5, R1, R2
           BRz      L1
           BRnzp    NEXT
L1         ADD      R1, R1, R1
           STR      R1, R0, #0
NEXT       ADD      R0, R0, #1
           BRnzp    LOOP
DONE       HALT
NUMBERS    .FILL    x4000
MASK       .FILL    x8000
 .END
```

7.16 Assume a sequence of nonnegative integers is stored in consecutive memory locations, one integer per memory location, starting at location x4000. Each integer has a value between 0 and 30,000 (decimal). The sequence terminates with the value −1 (i.e., xFFFF).

What does the following program do?

```
        .ORIG   x3000
        AND     R4, R4, #0
        AND     R3, R3, #0
        LD      R0, NUMBERS
LOOP    LDR     R1, R0, #0
        NOT     R2, R1
        BRz     DONE
        AND     R2, R1, #1
        BRz     L1
        ADD     R4, R4, #1
        BRnzp   NEXT
L1      ADD     R3, R3, #1
NEXT    ADD     R0, R0, #1
        BRnzp   LOOP
DONE    TRAP    x25
NUMBERS .FILL   x4000
        .END
```

7.17 Suppose you write two separate assembly language modules that you expect to be combined by the linker. Each module uses the label AGAIN, and neither module contains the pseudo-op `.EXTERNAL AGAIN`. Is there a problem using the label `AGAIN` in both modules? Why or why not?

7.18 The following LC-3 program compares two character strings of the same length. The source strings are in the `.STRINGZ` form. The first string starts at memory location x4000, and the second string starts at memory location x4100. If the strings are the same, the program terminates with the value 0 in R5. Insert instructions at (a), (b), and (c) that will complete the program.

```
        .ORIG   x3000
        LD      R1, FIRST
        LD      R2, SECOND
        AND     R0, R0, #0
LOOP    -------------- (a)
        LDR     R4, R2, #0
        BRz     NEXT
        ADD     R1, R1, #1
        ADD     R2, R2, #1
        -------------- (b)
        -------------- (c)
        ADD     R3, R3, R4
        BRz     LOOP
        AND     R5, R5, #0
        BRnzp   DONE
NEXT    AND     R5, R5, #0
        ADD     R5, R5, #1
DONE    TRAP    x25
FIRST   .FILL   x4000
SECOND  .FILL   x4100
        .END
```

7.19 When the following LC-3 program is executed, how many times will the instruction at the memory address labeled LOOP execute?

```
        .ORIG   x3005
        LEA     R2, DATA
        LDR     R4, R2, #0
LOOP    ADD     R4, R4, #-3
        BRzp    LOOPk                 TRAP    x25
DATA    .FILL   x000B
        .END
```

7.20 LC-3 assembly language modules (a) and (b) have been written by different programmers to store x0015 into memory location x4000. What is fundamentally different about their approaches?

a.

```
        .ORIG   x5000
        AND     R0, R0, #0
        ADD     R0, R0, #15
        ADD     R0, R0, #6
        STI     R0, PTR
        HALT
PTR     .FILL   x4000
        .END
```

b.

```
        .ORIG   x4000
        .FILL   x0015
        .END
```

7.21 Assemble the following LC-3 assembly language program.

```
        .ORIG   x3000
        AND     R0, R0, #0
        ADD     R2, R0, #10
        LD      R1, MASK
        LD      R3, PTR1
LOOP    LDR     R4, R3, #0
        AND     R4, R4, R1
        BRz     NEXT
        ADD     R0, R0, #1
NEXT    ADD     R3, R3, #1
        ADD     R2, R2, #-1
        BRp     LOOP
        STI     R0, PTR2
HALT
MASK    .FILL   x8000
PTR1    .FILL   x4000
PTR2    .FILL   x5000
        .END
```

What does the program do (in no more than 20 words)?

7.22 The LC-3 assembler must be able to map an instruction's mnemonic opcode into its binary opcode. For instance, given an ADD, it must generate the binary pattern 0001. Write an LC-3 assembly language program that prompts the user to type in an LC-3 assembly language opcode and then displays its binary opcode. If the assembly language opcode is invalid, it displays an error message.

7.23 The following LC-3 program determines whether a character string is a palindrome or not. A palindrome is a string that reads the same backwards as forwards. For example, the string "racecar" is a palindrome. Suppose a string starts at memory location x4000, and is in the `.STRINGZ` format. If the string is a palindrome, the program terminates with the value 1 in R5. If not, the program terminates with the value 0 in R5. Insert instructions at (a)–(e) that will complete the program.

```
        .ORIG   x3000
        LD      R0, PTR
        ADD     R1, R0, #0
AGAIN   LDR     R2, R1, #0
        BRz     CONT
        ADD     R1, R1, #1
        BRnzp   AGAIN
CONT    --------------(a)
LOOP    LDR     R3, R0, #0
        --------------(b)
        NOT     R4, R4
        ADD     R4, R4, #1
        ADD     R3, R3, R4
        BRnp    NO
        --------------(c)
        --------------(d)
        NOT     R2, R0
        ADD     R2, R2, #1
        ADD     R2, R1, R2
        BRnz    YES
        --------------(e)
YES     AND     R5, R5, #0
        ADD     R5, R5, #1
        BRnzp   DONE
NO      AND     R5, R5, #0
DONE    HALT
PTR     .FILL   x4000
        .END
```

7.24 We want the following program fragment to shift R3 to the left by four bits, but it has an error in it. Identify the error and explain how to fix it.

```
        .ORIG   x3000
        AND     R2, R2, #0
        ADD     R2, R2, #4
LOOP    BRz     DONE
        ADD     R2, R2, #-1
        ADD     R3, R3, R3
        BR      LOOP
DONE    HALT
        .END
```

7.25 What does the pseudo-op `.FILL xFF004` do? Why?