Chapter 7

Assembly Language

The following is provided as reference material to the Assembly process, and the LC-3b Assembly Language. It has been extracted from Intro to Computing Systems: From bits and gates to C and beyond, 2e, McGraw-Hill, 2004. In my urgency to get this on the web site, I may have inadvertently created inconsistencies. If you find anything here that is missing an antecedent or otherwise makes no sense, please contact me and/or one of the TAs. – Yale Patt

7.1 LC-3b Assembly Language

We will begin our study of the LC-3b assembly language by means of an example. The program in Figure 7.1 multiplies the positive integer initially stored in NUMBER by six 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, 0E, and 10) specify actual instructions to be translated into instructions in the ISA of the LC-3b, 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-3b assembler), and the translation process is called *assembly*.

7.1.1 Instructions

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

```
LABEL   OPCODE   OPERANDS   ; COMMENTS
```

```
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     LEA     R2,NUMBER
07          LDW     R2,R2,#0
08     LEA     R1,SIX
09          LDW     R1,R1,#0
0A          AND     R3,R3,#0      ; Clear R3. It will
0B                                ; contain the product.
0C  ; The inner loop
0D  ;
0E  AGAIN   ADD     R3,R3,R2
0F          ADD     R1,R1,#-1     ; R1 keeps track of
10          BRp     AGAIN         ; the iterations
11  ;
12          HALT
13  ;
14  NUMBER  .BLKW   1
15  SIX     .FILL   x0006
16  ;
17          .END
```

Figure 7.1: An assembly language program

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).

The OPCODE is a symbolic name for the opcode of the corresponding LC-3b instruction. The idea is that it is easier to remember an operation by the symbolic name ADD, AND, or LDW than by the four-bit quantity 0001,0101, or 0110.

The number of operands depends on the operation being performed. For example, the ADD instruction (line 0E) 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, R2, . . . , R7.

The LEA instruction (line 06) requires two operands (the memory location whose address is to be read) and the destination register which is to contain that address after the instruction completes execution. We will see momentarily that memory locations will be given symbolic addresses called *labels*. In this case, the location whose address is to be read is given the label *NUMBER*. The destination into which that address is to be loaded is register 2.

```
LEA   R2, NUMBER
```

As we discussed in class, 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 06 and SIX in line 08). In the case of immediate operands, the actual value is explicitly represented (such as the value 0 in line 0A).

```
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 which are used to identify memory locations that are referred to explicitly in the program. In LC-3b assembly language, a label consists of from one to 20 alphanumeric characters (i.e., a capital or lower case 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-3b 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 0E).

2. The location contains a value that is loaded or stored (for example, NUMBER, line 14, and SIX, line 15).

The location AGAIN is specifically referenced by the branch instruction in line 10.

```
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 LEA instruction in line 06. 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-3b 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 non-blank 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 line 0A, the comment "Clear R3; it will contain the product" lets the reader know that the instruction on line 0A is initializing R3 prior to accumulating the product of the two numbers. While the purpose of line 0A 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 to helping 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 0F, 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 0E through 10 are separated from the rest of the code by lines 0D and 11. There is nothing on lines 0D and 11 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.

7.1.2 Pseudo-ops (Assembler Directives)

The LC-3b assembler is a program that takes as input a string of characters representing a computer program written in LC-3b assembly language, and translates it into a program in the ISA of the LC-3b. 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-3b 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-3b program. In line 05, .ORIG x3050 says, start with location x3050. As a result, the LEA R2,NUMBER 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 15, the ninth location in the resultant LC-3b 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 Words) in the program. The actual number is the operand of the .BLKW pseudo-op. In line 11, 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 $n+a$ memory locations, where $a=1$ if n is odd, and $a=2$ if n is even. The argument is a sequence of n characters, inside double quotation marks. The first $n+1$ bytes of memory are initialized with the ASCII codes of the corresponding characters in the string, followed by x00. A final byte x00 is added if necessary to end the string on a word boundary. The $n+1st$ character (x00) 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: x48
x3011: x65
x3012: x6C
x3013: x6C
x3014: x6F
x3015: x2C
x3016: x20
x3017: x57
x3018: x6F
x3019: x72
x301A: x6C
x301B: x64
x301C: x21
x301D: x00
```

.END

.END tells the assembler where the program ends. Any characters that come after .END will not be utilized 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.

7.1.3 An Example

The program shown in Figure 7.2 takes a character that is input from the keyboard and a file and counts the number of occurrences of that character in that file.

```
;
; Program to count occurrences of a character in a File.
; Character to be input from the keyboard.
; Result to be displayed on the monitor.
; Program works only if no more than 9 occurrences are found.
;
;
; Initialization
;
        .ORIG   x3000
        AND     R2,R2,#0      ; R2 is counter, initialize to 0
        LEA     R3,PTR        ; R3 is pointer to characters
   LDW     R3,R3,#0
        TRAP    x23           ; R0 gets character input
        LDB     R1,R3,#0      ; R1 gets the next character
;
; Test character for end of file
;
TEST    ADD     R4,R1,#-4     ; Test for EOT
        BRz     OUTPUT        ; If done, prepare the output
;
; Test character for match.  If a match, increment count.
;
        NOT     R1,R1
        ADD     R1,R1,R0      ; If match, R1 = xFFFF
        NOT     R1,R1         ; If match, R1 = x0000
        BRnp    GETCHAR       ; If no match, do not increment
        ADD     R2,R2,#1
;
; Get next character from the file
;
GETCHAR ADD     R3,R3,#1      ; Increment the pointer
        LDB     R1,R3,#0      ; R1 gets the next character to test
        BRnzp   TEST
```

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

```
23   ;
24   ; Output the count.
25   ;
26   OUTPUT  LEA     R0,ASCII      ; Load the ASCII template
27           LDW     R0,R0,#0
28           ADD     R0,R0,R2      ; Convert binary to ASCII
29           TRAP    x21           ; ASCII code in R0 is displayed
2A           TRAP    x25           ; Halt machine
2B   ;
2C   ; Storage for pointer and ASCII template
2D   ;
2E   ASCII   .FILL   x0030
2F   PTR     .FILL   x4000
30           .END
```

Figure 7.2: The assembly language program to count occurrences of a character (continued)

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 0E). TRAP x21 causes the ASCII code in R0 to be displayed on the monitor (line 29). TRAP x25 causes the machine to be halted (line 2A).

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 2E 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 2F). 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.

7.2 The Assembly Process

Before an LC-3b 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-3b ISA. It is the job of the LC-3b assembler to perform that translation.

7.2.1 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 running example 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 attempt 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 a location x3000. The assembler then moves on to line 0B, which it can easily translate into LC-3b machine code. At this point, we have

```
x3000:  0101010010100000
```

The LC-3b 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 the above problem 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 one, we construct the symbol table. In pass two, 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

```
LEA R3,PTR
```

during the second pass, it already knows the correspondence between PTR and x3028 (from the first pass). Thus it can easily translate line 0C to

```
x3002:  1110011000010011
```

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

7.2.2 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 above 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.

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 sixth instruction in the program and the LC at that point contains x300A, a symbol table entry is constructed thus:

Symbol	Address
TEST	x300A

The second instruction that has a label is at line 20. At this point, the LC has been incremented to x3018. Thus a symbol table entry is constructed, as follows:

Symbol	Address
GETCHAR	x3018

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

Symbol	Address
TEST	x300A
GETCHAR	x3018
OUTPUT	x301E
ASCII	x3028
PTR	x302A

7.2.3 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-3b 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 x302A. The instruction is LEA, which has an opcode encoding of 1110. The Destination register (DR) is R3, that is, 011.

PCoffset is computed as follows: We know that PTR is the label for address x302A, and that the incremented PC is LC+2, in this case x3004. Since PTR (x302A) must be the sum of the incremented PC (x3004) and twice the sign-extended PCoffset (since the offset is in words and memory is byte-addressable), PCoffset must be x0013. Putting

Address	Binary
	0011000000000000
x3000	0101010010100000
x3002	1110011000010011
x3004	0110011011000000
x3006	1111000000100011
x3008	0010001011000000
x300A	0001100001111100
x300C	0000010000001000
x300E	1001001001111111
x3010	0001001001000000
x3012	1001001001111111
x3014	0000101000000001
x3016	0001010010100001
x3018	0001011011100001
x301A	0010001011000000
x301C	0000111111110110
x301E	1110000000000100
x3020	0110000000000000
x3022	0001000000000010
x3024	1111000000100001
x3026	1111000000100101
x3028	0000000000110000
x302A	0100000000000000

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

this all together, x3002 is set to 1110011000010011, and the LC is incremented to x3004.

Note: In order to use the LEA instruction, it is necessary that the source of the load, in this case the address whose label is PTR, is not more than +512 or -510 memory locations from the LEA instruction itself. If the address of PTR had been greater than LC+2 +510 or less than LC+2 -512, 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 completing successfully. Fortunately, PTR is close enough to the LEA 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-3b 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.

That process was, on a good day, merely tedious. Fortunately, you do not have to do it for a living—the LC-3b assembler does that. And, since you now know LC-

3b assembly language, there is no need to program in machine language. Now we can write our programs symbolically in LC-3b assembly language and invoke the LC-3b assembler to create the machine language versions that can execute on an LC-3b computer.