(a) Load 104

Step	RTN	PC	IR	MAR	MBR	AC
(initial values)		100	------	------	------	------
Fetch	MAR ← PC	100	------	100	------	------
	IR ← M[MAR]	100	1104	100	------	------
	PC ← PC + 1	101	1104	100	------	------
Decode	MAR ← IR[11–0]	101	1104	104	------	------
	(Decode IR[15–12])	101	1104	104	------	------
Get operand	MBR ← M[MAR]	101	1104	104	0023	------
Execute	AC ← MBR	101	1104	104	0023	0023

(b) Add 105

Step	RTN	PC	IR	MAR	MBR	AC
(initial values)		101	1104	104	0023	0023
Fetch	MAR ← PC	101	1104	101	0023	0023
	IR ← M[MAR]	101	3105	101	0023	0023
	PC ← PC + 1	102	3105	101	0023	0023
Decode	MAR ← IR[11–0]	102	3105	105	0023	0023
	(Decode IR[15–12])	102	3105	105	0023	0023
Get operand	MBR ← M[MAR]	102	3105	105	FFE9	0023
Execute	AC ← AC + MBR	102	3105	105	FFE9	000C

(c) Store 106

Step	RTN	PC	IR	MAR	MBR	AC
(initial values)		102	3105	105	FFE9	000C
Fetch	MAR ← PC	102	3105	102	FFE9	000C
	IR ← M[MAR]	102	2106	102	FFE9	000C
	PC ← PC + 1	103	2106	102	FFE9	000C
Decode	MAR ← IR[11–0]	103	2106	106	FFE9	000C
	(Decode IR[15–12])	103	2106	106	FFE9	000C
Get operand	(not necessary)	103	2106	106	FFE9	000C
Execute	MBR ← AC	103	2106	106	000C	000C
	M[MAR] ← MBR	103	2106	106	000C	000C

FIGURE 4.14 A Trace of the Program to Add Two Numbers

4.11 A DISCUSSION ON ASSEMBLERS

In the program shown in Table 4.3, it is a simple matter to convert from the assembly language instruction `Load 104`, for example, to the machine language instruction 1104 (in hex). But why bother with this conversion? Why not just write in machine code? Although it is very efficient for computers to see these instructions as binary numbers, it is difficult for human beings to understand and program in sequences of 0s and 1s. We prefer words and symbols over long numbers, so it seems a natural solution to devise a program that does this simple conversion for us. This program is called an **assembler**.

4.11.1 What Do Assemblers Do?

An assembler's job is to convert assembly language (using mnemonics) into machine language (which consists entirely of binary values, or strings of 0s and 1s). Assemblers take a programmer's assembly language program, which is really a symbolic representation of the binary numbers, and convert it into binary instructions, or the machine code equivalent. The assembler reads a **source file** (assembly program) and produces an **object file** (the machine code).

Substituting simple alphanumeric names for the opcodes makes programming much easier. We can also substitute **labels** (simple names) to identify or name particular memory addresses, making the task of writing assembly programs even simpler. For example, in our program to add two numbers, we can use labels to indicate the memory addresses, thus making it unnecessary to know the exact memory address of the operands for instructions. Table 4.4 illustrates this concept.

When the address field of an instruction is a label instead of an actual physical address, the assembler still must translate it into a real, physical address in main memory. Most assembly languages allow for labels. Assemblers typically specify formatting rules for their instructions, including those with labels. For example, a label might be limited to three characters and may also be required to occur as the first field in the instruction. MARIE requires labels to be followed by a comma.

Labels are nice for programmers. However, they make more work for the assembler. It must make two passes through a program to do the translation. This

Address		Instruction	
100		Load	X
101		Add	Y
102		Store	Z
103		Halt	
104	X,	0023	
105	Y,	FFE9	
106	Z,	0000	

TABLE 4.4 An Example Using Labels

means the assembler reads the program twice, from top to bottom each time. On the first pass, the assembler builds a set of correspondences called a **symbol table**. For the above example, it builds a table with three symbols: *X*, *Y*, and *Z*. Because an assembler goes through the code from top to bottom, it cannot translate the entire assembly language instruction into machine code in one pass; it does not know where the data portion of the instruction is located if it is given only a label. But after it has built the symbol table, it can make a second pass and "fill in the blanks."

In the above program, the first pass of the assembler creates the following symbol table:

X	104
Y	105
Z	106

It also begins to translate the instructions. After the first pass, the translated instructions would be incomplete as follows:

1			X
3			Y
2			Z
7	0	0	0

On the second pass, the assembler uses the symbol table to fill in the addresses and create the corresponding machine language instructions. Thus, on the second pass, it would know that *X* is located at address 104, and would then substitute 104 for the *X*. A similar procedure would replace the *Y* and *Z*, resulting in:

1	1	0	4
3	1	0	5
2	1	0	6
7	0	0	0

Because most people are uncomfortable reading hexadecimal, most assembly languages allow the data values stored in memory to be specified as binary, hexadecimal, or decimal. Typically, some sort of **assembler directive** (an instruction specifically for the assembler that is not supposed to be translated into machine code) is given to the assembler to specify which base is to be used to interpret the value. We use DEC for decimal and HEX for hexadecimal in MARIE's assembly language. For example, we rewrite the program in Table 4.4 as shown in Table 4.5.

Instead of requiring the actual binary data value (written in HEX), we specify a decimal value by using the directive DEC. The assembler recognizes this directive and converts the value accordingly before storing it in memory. Again, directives are not converted to machine language; they simply instruct the assembler in some way.

Another kind of directive common to virtually every programming language is the **comment delimiter**. Comment delimiters are special characters that tell the

Address		Instruction	
100		Load	X
101		Add	Y
102		Store	Z
103		Halt	
104	X,	DEC	35
105	Y,	DEC	-23
106	Z,	HEX	0000

TABLE 4.5 An Example Using Directives for Constants

assembler (or compiler) to ignore all text following the special character. MARIE's comment delimiter is a front slash ("/"), which causes all text between the delimiter and the end of the line to be ignored.

4.11.2 Why Use Assembly Language?

Our main objective in presenting MARIE's assembly language is to give you an idea of how the language relates to the architecture. Understanding how to program in assembly goes a long way toward understanding the architecture (and vice versa). Not only do you learn basic computer architecture, but you also can learn exactly how the processor works and gain significant insight into the particular architecture on which you are programming. There are many other situations where assembly programming is useful.

Most programmers agree that 10% of the code in a program uses approximately 90% of the CPU time. In time-critical applications, we often need to optimize this 10% of the code. Typically, the compiler handles this optimization for us. The compiler takes a high-level language (such as C++) and converts it into assembly language (which is then converted into machine code). Compilers have been around a long time and in most cases they do a great job. Occasionally, however, programmers must bypass some of the restrictions found in high-level languages and manipulate the assembly code themselves. By doing this, programmers can make the program more efficient in terms of time (and space). This hybrid approach (most of the program written in a high-level language, with part rewritten in assembly) allows the programmer to take advantage of the best of both worlds.

Are there situations in which entire programs should be written in assembly language? If the overall size of the program or response time is critical, assembly language often becomes the language of choice. This is because compilers tend to obscure information about the cost (in time) of various operations and programmers often find it difficult to judge exactly how their compiled programs will perform. Assembly language puts the programmer closer to the architecture and, thus, in firmer control. Assembly language might actually be necessary if the programmer wishes to accomplish certain operations not available in a high-level language.

A perfect example, in terms of both response performance and space-critical design, is found in **embedded systems**. These are systems in which the computer is integrated into a device that is typically not a computer. Embedded systems

must be reactive and often are found in time-constrained environments. These systems are designed to perform either a single instruction or a very specific set of instructions. Chances are you use some type of embedded system every day. Consumer electronics (such as cameras, camcorders, cellular phones, PDAs, and interactive games), consumer products (such as washers, microwave ovens, and washing machines), automobiles (particularly engine control and antilock brakes), medical instruments (such as CAT scanners and heart monitors), and industry (for process controllers and avionics) are just a few of the examples of where we find embedded systems.

The software for an embedded system is critical. An embedded software program must perform within very specific response parameters and is limited in the amount of space it can consume. These are perfect applications for assembly language programming. We delve deeper into this topic in Chapter 10.

4.12 EXTENDING OUR INSTRUCTION SET

Even though MARIE's instruction set is sufficient to write any program we wish, there are a few instructions we can add to make programming much simpler. We have 4 bits allocated to the opcode, which implies we can have 16 unique instructions, and we are using only 9 of them. We add the instructions from Table 4.6 to extend our instruction set.

The `JnS` (jump-and-store) instruction allows us to store a pointer to a return instruction and then proceeds to set the PC to a different instruction. This enables us to call procedures and other subroutines, and then return to the calling point in our code once the subroutine has finished. The `Clear` instruction moves all 0s into the accumulator. This saves the machine cycles that would otherwise be expended in loading a 0 operand from memory.

The `AddI` instruction (as well as the `JumpI` instruction) uses a different **addressing mode**. All previous instructions assume the value in the data portion of the instruction is the **direct address** of the operand required for the instruction. The `AddI` instruction uses the **indirect addressing mode** (we present more on addressing modes in Chapter 5). Instead of using the value found at location *X* as the actual address, we use the value found in *X* as a pointer to a new memory

Instruction Number (hex)	Instruction	Meaning
0	JnS *X*	Store the PC at address *X* and jump to *X* + 1.
A	Clear	Put all zeros in AC.
B	AddI *X*	Add indirect: Go to address *X*. Use the value at *X* as the actual address of the data operand to add to AC.
C	JumpI *X*	Jump indirect: Go to address *X*. Use the value at *X* as the actual address of the location to jump to.

TABLE 4.6 MARIE's Extended Instruction Set

location that contains the data we wish to use in the instruction. For example, if we have the instruction `AddI 400`, we would go to location 400, and assuming we found the value 240 stored at location 400, we would go to location 240 to get the actual operand for the instruction. We have, essentially, allowed for pointers in our language.

Returning to our discussion of register transfer notation, our new instructions are represented as follows:

JnS

```
MBR ← PC
MAR ← X
M[MAR] ← MBR
MBR ← X
AC ← 1
AC ← AC + MBR
PC ← AC
```

Clear

```
AC ← 0
```

AddI *X*

```
MAR ← X
MBR ← M[MAR]
MAR ← MBR
MBR ← M[MAR]
AC ← AC + MBR
```

JumpI *X*

```
MAR ← X
MBR ← M[MAR]
PC ← MBR
```

Table 4.7 summarizes MARIE's entire instruction set.

Let's look at some examples using the full instruction set.

EXAMPLE 4.1 Here is an example using a loop to add five numbers:

```
Address         Instruction
100             Load      Addr    /Load address of first number to be added
101             Store     Next    /Store this address as our Next pointer
102             Load      Num     /Load the number of items to be added
103             Subt      One     /Decrement
```

```
104             Store     Ctr     /Store this value in Ctr to control looping
105     Loop,   Load      Sum     /Load the Sum into AC
106             AddI      Next    /Add the value pointed to by location Next
107             Store     Sum     /Store this sum
108             Load      Next    /Load Next
109             Add       One     /Increment by one to point to next address
10A             Store     Next    /Store in our pointer Next
10B             Load      Ctr     /Load the loop control variable
10C             Subt      One     /Subtract one from the loop control variable
10D             Store     Ctr     /Store this new value in loop control variable
10E             Skipcond  000     /If control variable < 0, skip next
                                  /instruction
10F             Jump      Loop    /Otherwise, go to Loop
110             Halt              /Terminate program
111     Addr,   Hex       117     /Numbers to be summed start at location 118
112     Next,   Hex       0       /A pointer to the next number to add
113     Num,    Dec       5       /The number of values to add
114     Sum,    Dec       0       /The sum
115     Ctr,    Hex       0       /The loop control variable
116     One,    Dec       1       /Used to increment and decrement by 1
117             Dec       10      /The values to be added together
118             Dec       15
119             Dec       20
11A             Dec       25
11B             Dec       30
```

Note: Line numbers in program are given for information only and are not required for use in the MarieSim environment.

Although the comments are reasonably explanatory, let's walk through Example 4.1. Recall that the symbol table stores [label, location] pairs. The `Load Addr` instruction becomes `Load 111`, because `Addr` is located at physical memory address 111. The value of 117 (the value stored at `Addr`) is then stored in `Next`. This is the pointer that allows us to "step through" the five values we are adding (located at addresses 117, 118, 119, 11A, and 11B). The `Ctr` variable keeps track of how many iterations of the loop we have performed. Because we are checking to see if `Ctr` is negative to terminate the loop, we start by subtracting one from `Ctr`. `Sum` (with an initial value of 0) is then loaded in the AC. The loop begins, using `Next` as the address of the data we wish to add to the AC. The `Skipcond` statement terminates the loop when `Ctr` is negative by skipping the unconditional branch to the top of the loop. The program then terminates when the `Halt` statement is executed.

Example 4.2 shows how you can use the `Skipcond` and `Jump` instructions to perform selection. Although this example illustrates an `if/else` construct, you can easily modify this to perform an `if/then` structure, or even a `case` (or `switch`) structure.