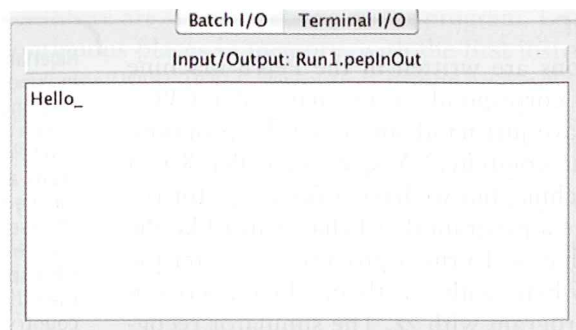
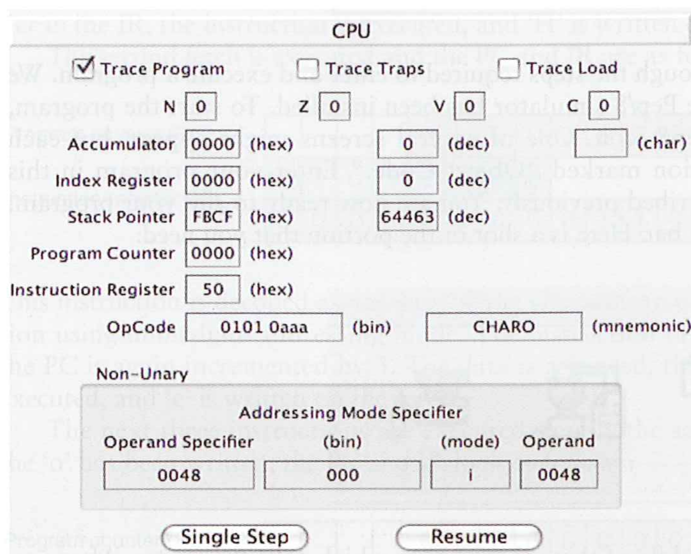


Pep/8 has a feature that lets you watch what is happening in the CPU as each instruction is executed. Here is a screen shot of the CPU after the program has been loaded. Notice that the "Trace Program" check box has been marked. This screen includes several boxes that we have not covered, but you can readily see the "Program Counter," "Instruction Register," and "OpCode" labels.

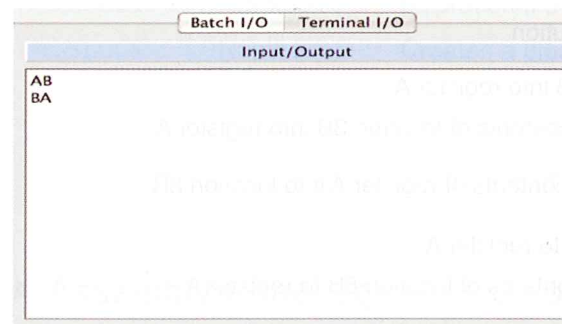


When the "Trace Program" option is checked, press the Single Step button and the first instruction will be executed. Continue pressing the Single Step button, and you can see the register values change.

Before we leave our machine code example, let's input two letters and print them out in reverse order. We can choose a place to put the input as it is read somewhere beyond the code. In this case we choose 0F and 12. We use direct addressing mode.

Action	Binary Instruction	Hex Instruction
Input a letter into location F	01001001 0000000000001000	49 000F
Input a letter into F + 1	01001001 0000000000010010	49 0010
Write out second letter	01010001 0000000000001000	51 0010
Write out first letter	01010001 0000000000001010	51 000F
Stop	00000000	00

Here is the object code and output window after entering 'A' and 'B':



6.4 Assembly Language

As we pointed out in Chapter 1, the first tools developed to help the programmer were assembly languages. **Assembly languages** assign mnemonic letter codes to each machine-language instruction. The programmer uses these letter codes in place of binary digits. The instructions in an assembly language are much like those we would use to tell someone how to do a calculation on a hand-held calculator.

Because every program that is executed on a computer eventually must be in the form of the computer's machine language, a program called an **assembler** reads each of the instructions in mnemonic form and translates

Assembly language A low-level programming language in which a mnemonic represents each of the machine-language instructions for a particular computer

Assembler A program that translates an assembly-language program in machine code

it into the machine-language equivalent. Also, because each type of computer has a different machine language, there are as many assembly languages and translators as there are types of machines.

Pep/8 Assembly Language

The goal of this section is not to make you become an assembly-language programmer, but rather to make you appreciate the advantages of assembly-language programming over machine coding. With this goal in mind, we cover only a few of Pep/8's assembly-language features here. We begin by examining the same operations we looked at in the last sections plus three other useful operations. In Pep/8's assembly language, there is a different opcode for each register, the operand is specified by "0x" and the hexadecimal value, and the addressing mode specifier is indicated by the letters 'i' or 'd'.

Mnemonic	Operand, Mode Specifier	Meaning of Instruction
STOP		Stop execution
LDA	0x008B,i	Load 008B into register A
LDA	0x008B,d	Load the contents of location 8B into register A
STA	0x008B,d	Store the contents of register A into location 8B
ADDA	0x008B,i	Add 008B to register A
ADDA	0x008B,d	Add the contents of location 8B to register A
SUBA	0x008B,i	Subtract 008B for register A
SUBA	0x008B,d	Subtract the contents of location 8B from register A
BR		Branch to the location specified in the operand specifier
CHARI	0x008B,d	Read a character and store it into location 8B
CHARO	0x008B,i	Write the character 8B
	0x000B,d	Write the character stored in location 0B
DECI	0x008B,d	Read a decimal number and store it into location 8B
DECO	0x008B,i	Write the decimal number 139 (8B in hex)
DECO	0x008B,d	Write the decimal number stored in location 8B

Did you wonder why we didn't do any arithmetic in machine language? Well, the output was defined only for characters. If we had done arithmetic, we would have had to convert the numbers to character form to see the results, and this is more complex than we wished to get. The Pep/8 assembly language provides the mnemonics DECI and DECO, which allow us to do decimal input and output. This terminology is somewhat misleading, however, because these operations actually involve calls to a series of instructions behind the scenes.

Assembler Directives

In a machine-language program, every instruction is stored in memory and then executed. Beginning with assembly languages, most programming languages have two kinds of instructions: instructions to be translated and instructions to the translating program. Here are a few useful **assembler directives** for the Pep/8 assembler—that is, instructions to the assembler. These instructions to the assembler are also called pseudo-operations.

Assembler directives Instructions to the translating program

Pseudo-op	Argument	Meaning
.ASCII	"Str\x00"	Represents a string of ASCII bytes
.BLOCK	Number of bytes	Creates a block of bytes
.WORD	Value	Creates a word and stores a value in it
.END		Signals the end of the assembly-language program

Assembly-Language Version of Program Hello

Let's take a look at the assembly-language program that writes "Hello" on the screen. Pep/8 assembly language allows us to directly specify the character to be output and to add a comment beside the instruction. A **comment** is text written for the human reader of the program that explains what is happening. Comments are an essential part of writing any program. The assembler ignores everything from the semicolon through the end of the line; it is a comment.

Comment Explanatory text for the human reader

```
CHARO  0x0048,i; Output an 'H'
CHARO  0x0065,i; Output an 'e'
CHARO  0x006C,i; Output an 'l'
CHARO  0x006C,i; Output an 'l'
CHARO  0x006F,i; Output an 'o'
STOP
.END
```

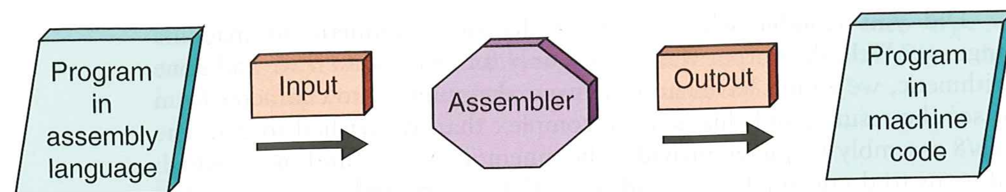


FIGURE 6.5 Assembly process

This code is entered into the Source Code window. The icon to the left of the load icon is the assembler icon. Click this icon, and the object code into which the program is translated appears in the Object Code window. The Assembler Listing window shows the address to which an instruction has been assigned, the object code, and the assembly-language code; it is shown here:

Assembler Listing

Addr	Code	Mnemon	Operand	Comment
0000	500048	CHARO	0x0048,i	
0003	500065	CHARO	0x0065,i	
0006	50006C	CHARO	0x006C,i	
0009	50006C	CHARO	0x006C,i	
000C	50006F	CHARO	0x006F,i	
000F	00	STOP		

The process of running a program coded in an assembly language is illustrated in Figure 6.5. The *input* to the assembler is a program written in assembly language. The *output* from the assembler is a program written in machine code. You can see why the creation of assembly language represented such an important step in the history of programming languages: It removed many of the details of machine-language programming by abstracting the instructions into words. Although it added a step to the process of executing a program (the translation of assembly to machine code), that extra step is well worth the effort to make the programmer's life easier.

A New Program

Let's make a step up in complexity and write a program to read in three numbers and write out their sum. How would we do this task by hand? If we had a calculator, we would first clear the total; that is, we would set the sum to zero. Then we would get the first number and add it to the total, get the second number and add it to the total, and finally get the third number and add it to the total. The result would be what is in the accumulator of the calculator. We can model our computer program on this by-hand solution.

The most complex problem is that that we must associate four identifiers with places in memory, and this requires knowing how many places the program itself takes—that is, if we put the data at the end of the program. Let's make this process easier by putting our data before the program. We can start associating identifiers with memory locations beginning with location 0001 and have the fetch–execute cycle skip over these places to continue with the program. In fact, we can assign identifiers to the memory locations and use these names later in the program. We set up space for the sum using the `.WORD` pseudo-op so that we can set the contents to 0. We set up space for the three numbers using the `.BLOCK` pseudo-op.

```
      BR        main     ; Branch around data
sum:  .WORD     0x0000   ; Set up word with zero
num1: .BLOCK 2           ; Set up a two byte block for num1
num2: .BLOCK 2           ; Set up a two byte block for num2
num3: .BLOCK 2           ; Set up a two byte block for num3

main: LDA       sum,d    ; Load zero into the accumulator
      DECI      num1,d   ; Read and store num1
      ADDA      num1,d   ; Add num1 to accumulator
      DECI      num2,d   ; Read and store num2
      ADDA      num2,d   ; Add num2 to accumulator
      DECI      num3,d   ; Read and store num3
      ADDA      num3,d   ; Add num3 to accumulator
      STA       sum,d    ; Store accumulator into sum
      DECO      sum,d    ; Output sum
      STOP               ; Stop the processing
      .END               ; End of the program
```

Here is the assembler listing for this program, followed by the Input/Output window after we execute the program. Note that the user keys in the three values, and the program prints their sum.

Assembler Listing

```
Addr  Code    Symbol  Mnemon  Operand   Comment
0000  04000B          BR      main      ; Branch around data
0003  0000    sum:    .WORD   0x0000    ; Set up word with zero
0005  0000    num1:   .BLOCK  2         ; Set up a two byte block for num1
0007  0000    num2:   .BLOCK  2         ; Set up a two byte block for num2
0009  0000    num3:   .BLOCK  2         ; Set up a two byte block for num3

000B  C10003  main:   LDA     sum,d     ; Load zero into the accumulator
000E  310005          DECI    num1,d    ; Read and store num1
0011  710005          ADDA    num1,d    ; Add num1 to accumulator
0014  310007          DECI    num2,d    ; Read and store num2
0017  710007          ADDA    num2,d    ; Add num2 to accumulator
001A  310009          DECI    num3,d    ; Read and store num3
001D  710009          ADDA    num3,d    ; Add num3 to accumulator
0020  E10003          STA     sum,d     ; Store accumulator into sum
0023  390003          DECO    sum,d     ; Output sum
0026  00              STOP              ; Stop the processing
                      .END              ; End of the program

Symbol  Value
sum     0003
num1    0005
num2    0007
num3    0009
main    000B
```

Input/Output

```
23
14
2
39
```

A Program with Branching

We have shown that the program counter can be changed with a BR instruction that sets the program counter to the address of an instruction to execute next. Are there other ways to change the flow of control of the program? Can we ask a question and take one or another action on the basis of the answer to our question? Sure—let's see how. Here are two useful opcodes and their meaning:

Mnemonic	Operand, Mode Specifier	Meaning of Instruction
BRLT	i	Set the PC to the operand if the A register is less than zero
BREQ	i	Set the PC to the operand if the A register is equal to zero

For example:

```
LDA  num1,d      ; Load num1 into A register
BRLT lessThan    ; Branch to lessThan if num1 is less than 0
```

If the value stored in `num1` is negative when it is loaded into the A register, the PC is set to location `lessThan`. If the value is not negative, the PC is unchanged.

Let's change the previous program so that it prints the sum if it is positive and displays an error message if the sum is negative. Where should the test go? Just before the contents of the answer is stored into location sum, we can test the A register and print 'E' if it is negative.

We can use the BRLT instruction to test whether the sum is negative. If the A register is negative, the operand beside the BRLT instruction replaces the contents of the program counter so that the next instruction comes from there. We need to give the instruction a name, so we can branch to it. Let's call the instruction that prints the error message `negMsg`. When the error message has been written, we must branch back to the line that says STOP, which means we must name that line. Let's name it `finish`.

Here is the source code of this changed program. Note that we have reduced the number of comments. If the comment just duplicates the instruction, it can be a distraction instead of a help.

```
        BR      main        ; Branch around data
sum:    .WORD   0x0000      ; Set up word with zero
num1:   .BLOCK  2           ; Set up a two byte block for num1
num2:   .BLOCK  2           ; Set up a two byte block for num2
num3:   .BLOCK  2           ; Set up a two byte block for num3

negMsg: CHARO   0x0045,i    ; Print 'E'
        BR      finish      ; Branch to STOP instruction
main:   LDA     sum,d       ; Load zero into the accumulator
        DECI    num1,d      ; Read and add three numbers
        ADDA    num1,d
        DECI    num2,d
        ADDA    num2,d
        DECI    num3,d
        ADDA    num3,d
        BRLT    negMsg      ; Branch to negMsg of A < 0
        STA     sum,d       ; Store result into sum
        DECO    sum,d       ; Output sum
finish: STOP                ;
        .END                ;
```

Here is the assembler listing, followed by a screen shot of the input and output:

Assembler Listing

```
Addr  Code    Symbol   Mnemon  Operand    Comment
0000  040011           BR      main       ; Branch around data
0003  0000    sum:     .WORD   0x0000     ; Set up word with zero
0005  0000    num1:    .BLOCK  2          ; Set up a two byte block for num1
0007  0000    num2:    .BLOCK  2          ; Set up a two byte block for num2
0009  0000    num3:    .BLOCK  2          ; Set up a two byte block for num3
000B  500045  negMsg:  CHARO   0x0045,i   ; Print 'E'
000E  04002F           BR      finish     ; Branch to STOP instruction
0011  C10003  main:    LDA     sum,d      ; Load zero into the accumulator

0014  310005           DECI    num1,d     ; Read and add three numbers
0017  710005           ADDA    num1,d
001A  310007           DECI    num2,d
001D  710007           ADDA    num2,d
0020  310009           DECI    num3,d
0023  710009           ADDA    num3,d
0026  08000B           BRLT    negMsg     ; Branch to negMsg of A < 0
0029  E10003           STA     sum,d      ; Store result into sum
002C  390003           DECO    sum,d      ; Output sum
002F  00      finish:  STOP               ;
                       .END               ;

Symbol    Value
sum       0003
num1      0005
num2      0007
num3      0009
negMsg    000B
main      0011
finish    002F
```

Batch I/O | Terminal I/O

Input/Output

```
2
-3
-1
E
```


A Program with a Loop

What if we wanted to read and sum four values? Five values? Any number of values? We can input how many values we want to sum (`limit`) and write the code to read and sum that many values. We do so by creating a counting loop—a section of code that repeats a specified number of times. Within the code of the loop, a value is read and summed. How can we keep track of how many values we have read? We can make a hash mark each time we repeat the loop and compare the sum of the hash marks to the number of times we wish to repeat the loop. Actually, our hash mark is a place in memory where we store a 0; let's call it `counter`. Each time the loop is repeated, we add a 1 to that place in memory. When `counter` equals `limit`, we are finished with the reading and counting.

In the next section we describe pseudocode, a less wordy way of explaining what we do in branching and looping situations. For now, here is the code that reads in the number of data values to read and sum, reads and sums them, and prints the result:

```
          BR      main        ; Branch around data
sum:      .WORD   0x0000      ; Set up word with zero
num:      .BLOCK  2           ; Set up a block for num
limit:    .BLOCK  2           ; Set up a block for limit
counter:  .WORD   0x0000      ; Set up counter

main:     DECI    limit,d     ; Input limit
loop:     DECI    num,d       ; Read and sum limit numbers
          LDA     num,d
          ADDA    sum,d
          STA     sum,d
          LDA     counter,d   ; Load counter into A register
          ADDA    1,i         ; Add one to counter
          STA     counter,d
          CPA     limit,d     ; Compare counter and limit
          BREQ    quit        ; Go to quit if equal
          BR      loop        ; Repeat loop
quit:     DECO    sum,d       ; Output sum
          STOP
          .END
```

Here is the assembler listing, followed by a screen shot of a run:

```
Assembler Listing
Addr Code   Symbol   Mnemon Operand   Comment
0000 04000B          BR     main      ; Branch around data

0003 0000   sum:     .WORD  0x0000    ; Set up word with zero

0005 0000   num:     .BLOCK 2         ; Set up a block for num
0007 0000   limit:   .BLOCK 2
            ; Set up a block for limit
0009 0000   counter: .WORD  0x0000    ; Set up counter

000B 310007 main:    DECI   limit,d   ; Input limit
000E 310005 loop:    DECI   num,d     ; Read and sum limit numbers
0011 C10005          LDA    num,d
0014 710003          ADDA   sum,d
0017 E10003          STA    sum,d
001A C10009          LDA    counter,d ; Load counter into A register
001D 700001          ADDA   1,i       ; Add one to counter
0020 E10009          STA    counter,d
0023 B10007          CPA    limit,d   ; Compare counter and limit
0026 0A002C          BREQ   quit      ; Go to quit if equal
0029 04000E          BR     loop      ; Repeat loop
002C 390003 quit:    DECO   sum,d     ; Output sum
002F 00              STOP
                     .END

Symbol   Value
sum      0003
num      0005
limit    0007
counter  0009
main     000B
loop     000E
quit     002C

No errors. Successful assembly.
```

```
Batch I/O   Terminal I/O
Input/Output: loopWlimit.pepInOut
3
2
4
6
12
```


6.5 Expressing Algorithms

In the previous sections, we have written programs to write out a greeting, read numbers in and write them out in reverse order, add three numbers together and print an error message if the sum is negative, and enter a value and read and sum that many numbers. We expressed the solution to each problem in paragraph form and then wrote the code. In computing, the plan for a solution is called an **algorithm**. As you saw, going from the problem in paragraph form to the code is not always a clear-cut process. **Pseudocode** is a language that allows us to express algorithms in a clearer form.

The Music Genome Project
In 2002, Will Glaser, Jon Craft, and Tim Westergren founded the company Savage Beast Technologies and created the Music Genome Project. The project, which was created to capture "the essence of music at the most fundamental level," uses hundreds of musical attributes or "genes" to describe scores, as well as an intricate mathematical formula to analyze them. The project has analyzed tens of thousands of diverse scores and artists for attributes such as melody, harmony and rhythm, instrumentation, orchestration, arrangement, and lyrics. The results are incorporated in a program named Pandora and used to provide music recommendations for users listening on the Internet. When a user enters a song title into Pandora's search function, Pandora scans all of its analyzed song information to provide the most accurate and enjoyable playlist possible.

Pseudocode Functionality

In Chapter 1, we talked about the layers of language surrounding the actual machine. We didn't mention pseudocode at that time because it is not a computer language, but rather a shorthand-like language that people use to express actions. There are no special grammar rules for pseudocode, but to express actions we must be able to represent the following concepts.

Variables

Names that appear in pseudocode algorithms refer to places in memory where values are stored. The name should reflect the role of the content in the algorithm.

Algorithm A plan or outline of a solution; a logical sequence of steps that solve a problem

Pseudocode A language designed to express algorithms

Assignment

If we have variables, we must have a way to put a value into one. We use the statement

```
Set sum to 0
```

to store a value into the variable sum. Another way of expressing the same concept uses a back arrow (<—):

```
sum <— 1
```

If we assign values to variables with the assignment statement, how do we access them later? We access values in sum and num in the following statement:

```
Set sum to sum + num
```

or

```
sum <— sum + num
```