6.1 Computer Operations

The programming languages we use must mirror the types of operations that a computer can perform. So let's begin our discussion by repeating the definition of a computer: A computer is a programmable electronic device that can store, retrieve, and process data.

The operational words here are *programmable*, *store*, *retrieve*, and *process*. In a previous chapter we pointed out the importance of the realization that data and instructions to manipulate the data are logically the same and could be stored in the same place. That is what the word *programmable* means in this context. The instructions that manipulate data are stored within the machine along with the data. To change what the computer does to the data, we change the instructions.

Store, *retrieve*, and *process* are actions that the computer can perform on data. That is, the instructions that the control unit executes can store data into the memory of the machine, retrieve data from the memory of the machine, and process the data in some way in the arithmetic/logic unit. The word *process* is very general. At the machine level, processing involves performing arithmetic and logical operations on data values.

Where does the data that gets stored in the computer memory come from? How does the human ever get to see what is stored there, such as the results of some calculation? There are other instructions that specify the interaction between an input device and the CPU and between the CPU and an output device.

6.2 Machine Language

Machine language The language made up of binary-coded instructions that is used directly by the computer

As we pointed out in Chapter 1, the only programming instructions that a computer actually carries out are those written using **machine language**, the instructions built into the hardware of a particular computer. Initially humans had no choice except to write programs in machine language because other programming languages had not yet been invented.

So how are computer instructions represented? Recall that every processor type has its own set of specific machine instructions. These are the only instructions the processor can actually carry out. Because a finite number of instructions exist, the processor designers simply list the instructions and assign them a binary code that is used to represent them. This is similar to the approach taken when representing character data, as described in Chapter 3.

The relationship between the processor and the instructions it can carry out is completely integrated. The electronics of the CPU inherently recognize the binary representations of the specific commands, so there is no actual list of commands the computer must consult. Instead, the CPU embodies the list in its design.

Each machine-language instruction performs only one very low-level task. Each small step in a process must be explicitly coded in machine language. Even the small task of adding two numbers together must be broken down into smaller steps: enter a number into the accumulator, add a number to it, save the result. Then these three instructions must be written in binary, and the programmer has to remember which combination of binary digits corresponds to which instruction. As we mentioned in Chapter 1, machine-language programmers have to be very good with numbers and very detail oriented.

However, we can't leave you with the impression that only mathematicians can write programs in machine language. It is true that very few programs are written in machine language today, primarily because they represent an inefficient use of a programmer's time. Although most programs are written in higher-level languages and then translated into machine language (a process we describe later in this chapter), every piece of software is actually implemented in machine code. Understanding even just a little about this level will make you a more informed user. In addition, this experience emphasizes the basic definition of a computer and makes you appreciate the ease with which people interact with computers today.

Managing endangered species
Zoos have established captive populations of endangered animals to save them from extinction, but they need to have a good distribution of ages and genetic diversity to protect the species against diseases and inbreeding. A computerized database of all captive animals that contains dates of births and deaths, gender, parentage, and location enables scientists to measure important factors governing the welfare of a species, such as reproductive and survival rates, degree of inbreeding, and loss of genetic diversity. For example, the Minnesota Zoological Garden coordinates the International Species Inventory System (ISIS). ISIS provides global information on many different species of animals, including more than 163,000 living animals, and many endangered animals are being bred in captivity due to its help.

Pep/8: A Virtual Computer

By definition, machine code differs from machine to machine. Recall that just as each lock has a specific key that opens it, each type of computer has a specific set of operations that it can execute, called the computer's machine language. That is, each type of CPU has its own machine language that it understands. So how can we give each of you the experience of using machine language when you may be working on different machines? We solve that problem by using a **virtual computer**. A virtual computer is a hypothetical machine—in this case, one that is designed to contain the important features of real computers that we want to illustrate. Pep/8, designed by Stanley Warford, is the virtual machine that we use here.[1]

Virtual computer (machine) A hypothetical machine designed to illustrate important features of a real machine

Pep/8 has 39 machine-language instructions. This means that a program for Pep/8 must be a sequence consisting of a combination of these instructions. Don't panic: We will not ask you to understand and remember 39 sequences of binary bits! We merely plan to examine a few of these instructions, and we will not ask you to memorize any of them.

Important Features Reflected in Pep/8

The memory unit of the Pep/8 is made up of 65,536 bytes of storage. The bytes are numbered from 0 through 65,535 (decimal). Recall that each byte contains 8 bits, so we can describe the bit pattern in a byte using 2 hexadecimal digits. (Refer to Chapter 2 for more information on hexadecimal digits.) The word length in Pep/8 is 2 bytes, or 16 bits. Thus the information that flows into and out of the arithmetic/logic unit (ALU) is 16 bits in length.

Recall from Chapter 5 that a register is a small area of storage in the ALU of the CPU that holds special data and intermediate values. Pep/8 has seven registers, three of which we focus on at this point:

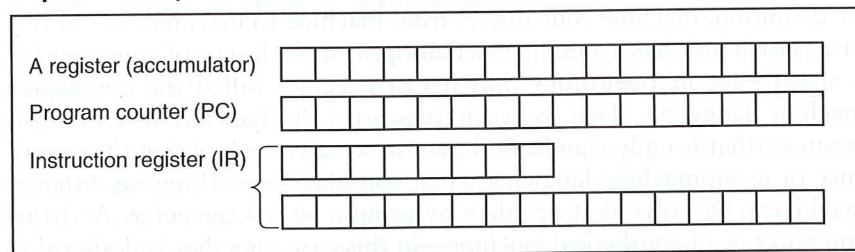
- The program counter (PC), which contains the address of the next instruction to be executed
- The instruction register (IR), which contains a copy of the instruction being executed
- The accumulator (A register)

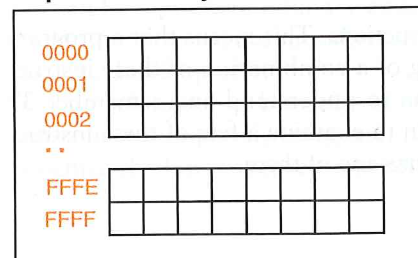
The accumulator is used to hold data and the results of operations; it is the special storage register referred in Chapter 5 in the discussion of the ALU.

We realize that this is a lot of detailed information, but don't despair! Remember that our goal is to give you a feel for what is actually happening at the lowest level of computer processing. By necessity, that processing keeps track of many details.

Figure 6.1 shows a diagram of Pep8's CPU and memory. Notice that the addresses in memory appear in orange. This color is intended to

FIGURE 6.1 Pep/8's architecture





emphasize that the addresses themselves are not stored in memory, but rather that they *name* the individual bytes of memory. We refer to any particular byte in memory by its address.

Before we go on, let's review some aspects of binary and hexadecimal numbers. The largest decimal value that can be represented in a byte is 255. It occurs when all of the bits are 1s: 11111111 in binary is FF in hexadecimal and 255 in decimal. The largest decimal value that can be represented in a word (16 bits) is 65,535. It occurs when all 16 bits are 1s: 1111111111111111 in binary is FFFF in hexadecimal and 65,535 in decimal. If we represent both positive and negative numbers, we lose a bit in the magnitude (because one is used for the sign), so we can represent values ranging from −7FFF to +7FFF in hexadecimal, or −32,767 to +32,767 in decimal.

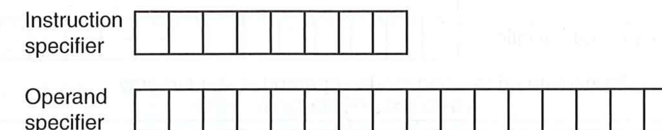
This information is important when working with the Pep/8 machine. The number of bits we have available determines the size of the numbers we can work with.

Instruction Format

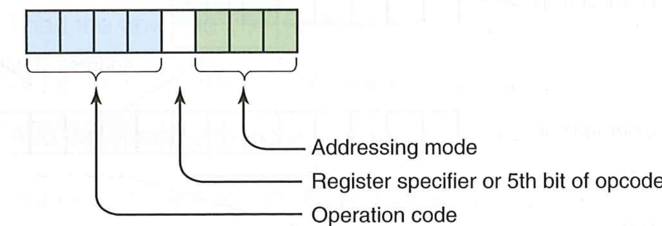
We have talked about instructions going into the instruction register, being decoded, and being executed. Now we are ready to look at a set (or subset) of concrete instructions that a computer can execute. First, however, we need to examine the format of an instruction in Pep/8.

Figure 6.2(a) shows the format for an instruction in Pep/8. There are two parts to an instruction: the *instruction specifier* and (optionally) the 16-bit *operand specifier*. The instruction specifier indicates which operation is to be carried out, such as "add a number to a value already stored in a register," and how to interpret just where the operand is. The operand

FIGURE 6.2 Pep/8 instruction format



(a) The two parts of an instruction



(b) The instruction specifier part of an instruction

specifier (the second and third bytes of the instruction) holds either the operand itself or the address of where the operand is to be found. Some instructions do not use the operand specifier.

The format of the instruction specifier varies depending on the number of bits used to represent a particular operation. In Pep/8, *operation codes* (called opcodes) vary from 4 bits to 8 bits long. The opcodes that we cover are 4 or 5 bits long, with the fifth bit of 4-bit opcodes used to specify which register to use. The *register specifier* is 0 for register A (the accumulator), which is the only register that we will use. Thus the register specifier is only color coded in our diagrams when it is part of the opcode. [See Figure 6.2(b).]

The 3-bit *addressing mode specifier* (shaded green) indicates how to interpret the operand part of the instruction. If the addressing mode is 000, the operand is in the operand specifier of the instruction. This addressing mode is called immediate (i). If the addressing mode is 001, the operand is the memory address named in the operand specifier. This addressing mode is called direct (d). (Other addressing modes also exist, but we do not cover them here.) The distinction between the immediate addressing mode and the direct addressing mode is very important because it determines where the data involved in the operation is stored or is to be stored. See Figure 6.3. Locations that contain addresses are shaded in orange; operands are shaded in gray.

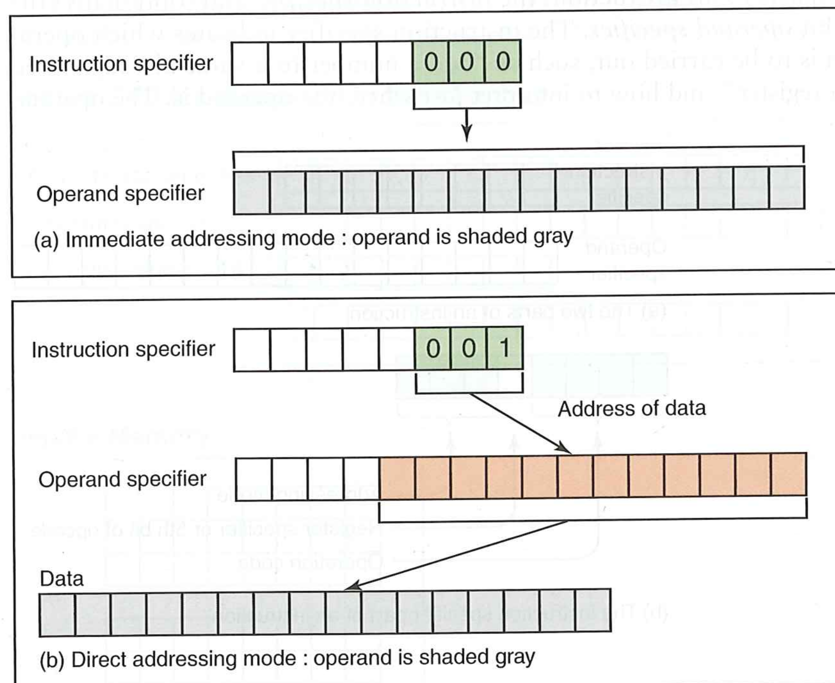


FIGURE 6.3 Difference between immediate addressing mode and direct addressing mode

Instructions that do not have an operand (data to be manipulated) are called *unary instructions*; they do not have an operand specifier. That is, unary instructions are only 1 byte long rather than 3 bytes long.

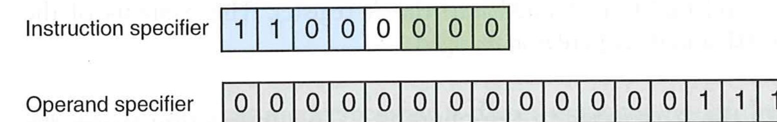
Some Sample Instructions

Let's look at some specific instructions in isolation and then put them together to write a program. Figure 6.4 contains the 4-bit operation code (or opcode) for the operations we are covering.

0000 Stop execution During the fetch–execute cycle, when the operation code is all zeros, the program halts. Stop is a unary instruction, so it occupies only one byte. The three rightmost bits in the byte are ignored.

1100 Load the operand into the A register This instruction loads one word (two bytes) into the A register. The mode specifier determines where the word is located. Thus the load opcode has different meanings depending on the addressing mode specifier. The mode specifier determines whether the value to be loaded is in the operand part of the instruction (the second and third bytes of the instruction) or is in the place named in the operand.

Let's look at concrete examples of each of these combinations. Here is the first 3-byte instruction:



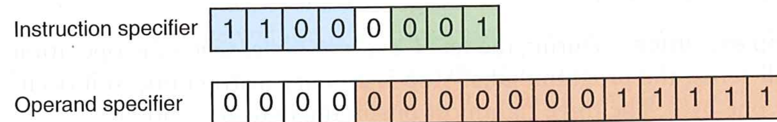
The addressing mode is immediate, meaning that the value to be loaded into the A register is in the operand specifier. That is, the data is in the

Opcode	Meaning of Instruction
0000	Stop execution
1100	Load the operand into the A register
1110	Store the contents of the A register into the operand
0111	Add the operand to the A register
1000	Subtract the operand to the A register
01001	Character input to the operand
01010	Character output from the operand

FIGURE 6.4 Subset of Pep/8 instructions

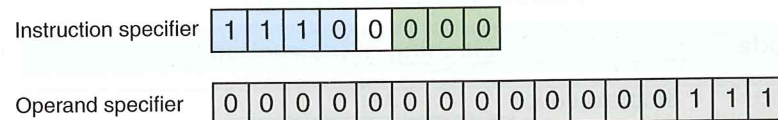
operand specifier, so it is shaded gray. After execution of this instruction, the contents of the second and third bytes of the instruction (the operand specifier) would be loaded into the A register (the accumulator). That is, the A register would contain 0007 and the original contents of A would be lost.

Here is another load instruction:

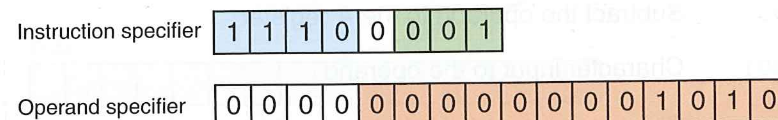


The addressing mode is direct, which means that the operand itself is not in the operand specifier (second and third bytes of the instruction); instead, the operand specifier holds the *address* (orange) of where the operand resides in memory. Thus, when this instruction is executed, the contents of *location* 001F would be loaded into the A register. Note that we have shaded the bits that represent a memory address in orange just as we have used orange for other addresses. The A register holds a word (2 bytes), so when an address is used to specify a word (rather than a single byte) as in this case, the address given is of the leftmost byte in the word. Thus the contents of adjacent locations 001F and 0020 are loaded into the A register. The contents of the operand (001F and 0020) are not changed.

1110 Store the A register to the operand This instruction stores the contents of the A register into the location specified in the operand, which is either the operand itself or the place named in the operand.

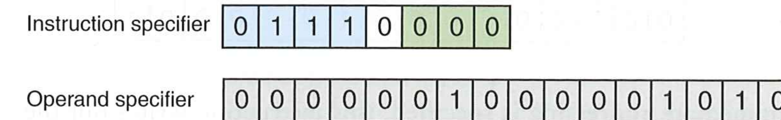


This instruction stores the value in the A register into the operand specifier of the instruction itself. The operand is gray to indicate that it consists of data.

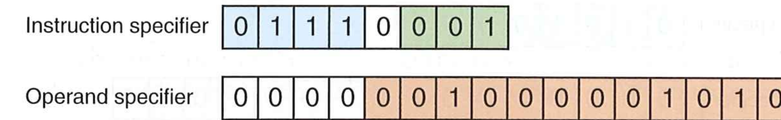


This instruction stores the contents of the A register into the word beginning at location 000A. It is invalid to use an immediate addressing mode with a store opcode; that is, we cannot try to store the contents of a register into the operand specifier.

0111 Add the operand to the A register Like the load operation, the add operation uses the addressing mode specifier, giving alternative interpretations. The two alternatives for this instruction are shown below with the explanation following each instruction.



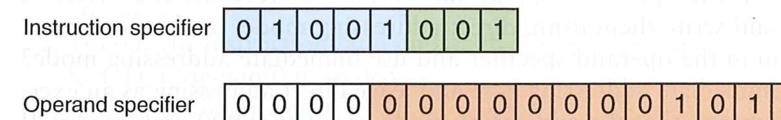
The contents of the second and third bytes of the instruction (the operand specifier) are added to the contents of the A register (20A in hex). Thus we have shaded the operand specifier to show that it is data.



Because the address mode specifier is direct, the contents of the operand specified in the second and third bytes of the instruction (location 020A) are added into the A register.

1000 Subtract the operand This instruction is just like the add operation except that the operand is subtracted from the A register rather than added. As with the load and add operations, there are variations of this instruction depending on the addressing mode.

0100 Character input to the operand This instruction allows the program to enter an ASCII character from the input device while the program is running. Only direct addressing is allowed, so the character is stored in the address shown in the operand specifier.



This instruction reads an ASCII character from the input device and stores it into location 000A.

0101 Character output from the operand This instruction sends an ASCII character to the output device while the program is running. The addressing mode may be either immediate or direct.

Instruction specifier	0	1	0	1	0	0	0	0

Operand specifier	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	1

Because immediate addressing is specified, this instruction writes out the ASCII character stored in the operand specifier. The operand specifier contains 1000001, which is 41 in hex and 65 in decimal. The ASCII character corresponding to that value is 'A', so the letter A is written to the screen.

Instruction specifier	0	1	0	1	0	0	0	1

Operand specifier	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1	0

Because direct addressing is used, this instruction writes out the ASCII character stored in the location named in the operand specifier, location 000A. What is written? We cannot say unless we know the contents of 000A. The ASCII character corresponding to whatever is stored at location 000A is printed.

6.3 A Program Example

We are now ready to write our first machine-language program: Let's write "Hello" on the screen. There are six instructions in this program: five to write out a character and one to indicate the end of the process. The instruction to write a character on the screen is 0101, the "Character output from the operand" operation. Should we store the characters in memory and write them using direct addressing mode, or should we just store them in the operand specifier and use immediate addressing mode? Let's use immediate addressing here and leave direct addressing as an exercise. This means that the addressing mode specifier is 000 and the ASCII code for the letter goes into the third byte of the instruction.

Action	Binary Instruction	Hex Instruction
Write "H"	01010000 0000000001001000	50 0048
Write "e"	01010000 0000000001100101	50 0065
Write "l"	01010000 0000000001101100	50 006C
Write "l"	01010000 0000000001101100	50 006C
Write "o"	01010000 0000000001101111	50 006F
Stop	00000000	00

The machine-language program is shown in binary in the second column and in hexadecimal in the third column. We must construct the operation specifier in binary because it is made up of a 4-bit opcode, a 1-bit register specifier, and a 3-bit addressing mode specifier. Once we have the complete 8 bits, we can convert the instruction to hexadecimal. Alternatively, we could construct the operand specifier directly in hexadecimal.

We used double quotes when referring to a collection of characters like "Hello" and single quotes when referring to a single character. This pattern is commonly used in programming languages, so we follow this convention here.

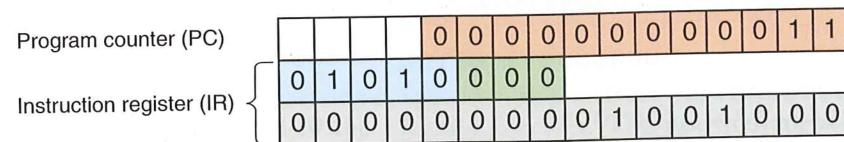
Hand Simulation

Let's simulate this program's execution by following the steps of the fetch–execute cycle. Such traces by hand really drive home the steps that the computer carries out.

Recall the four steps in the fetch–execute cycle:

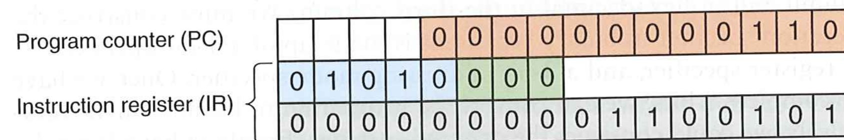
1. Fetch the next instruction (from the place named in the program counter).
2. Decode the instruction (and update the program counter).
3. Get data (operand) if needed.
4. Execute the instruction.

There are six instructions in our program. Let's assume that they are in contiguous places in memory, with the first instruction stored in memory locations 0000–0002. Execution begins by loading 0000 into the program counter (PC). At each stage of execution, let's examine the PC (shown in yellow) and the instruction register (IR). The program does not access the A register, so we do not bother to show it. At the end of the first fetch, the PC and the IR look like the following diagram. (We continue to use color to emphasize the addresses, opcode, address mode specifier, and data.) Notice that the program counter is incremented as soon as the instruction has been accessed.



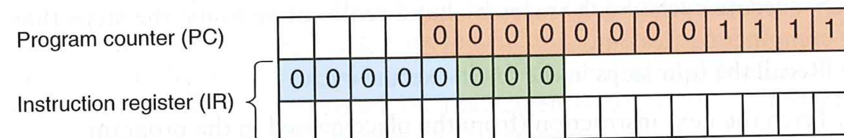
This instruction is decoded as a "Write character to output" instruction using immediate addressing mode. Because this instruction takes 3 bytes, the PC is incremented by 3. The data is retrieved from the operand specifier in the IR, the instruction is executed, and 'H' is written on the screen.

The second fetch is executed and the PC and IR are as follows:



This instruction is decoded as another "Write character to output" instruction using immediate addressing mode. The instruction takes 3 bytes, so the PC is again incremented by 3. The data is retrieved, the instruction is executed, and 'e' is written on the screen.

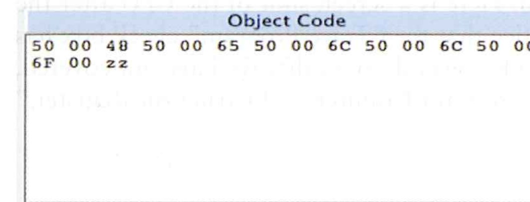
The next three instructions are executed exactly the same way. After the 'o' has been written, the PC and IR look as follows:



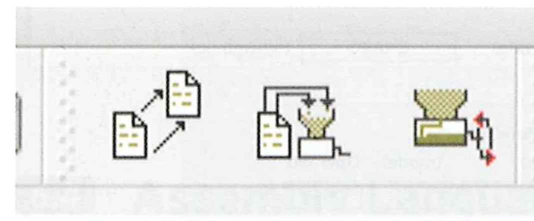
The opcode is decoded as a "Stop" instruction, so the contents of the addressing mode and the operand specifier are ignored. At this point, the fetch–execute cycle stops.

■ Pep/8 Simulator

Recall that the instructions are written in the Pep/8 machine language, which doesn't correspond to any particular CPU's machine language. We have just hand simulated the program. Can we execute it on the computer? Yes, we can. Pep/8 is a virtual (hypothetical) machine, but we have a *simulator* for the machine. That is, we have a program that behaves just like the Pep/8 virtual machine behaves. To run a program, we enter the hexadecimal code byte by byte, with exactly one blank between each byte, and end the program with zz. The simulator recognizes two z's as the end of the program. Here is a screen shot of the Pep/8 machine-language program:



Let's go through the steps required to enter and execute a program. We assume that the Pep/8 simulator has been installed. To start the program, click on the Pep8 icon. One of several screens might appear, but each contains a section marked "Object Code." Enter your program in this window as described previously. You are now ready to run your program. Go to the menu bar. Here is a shot of the portion that you need:

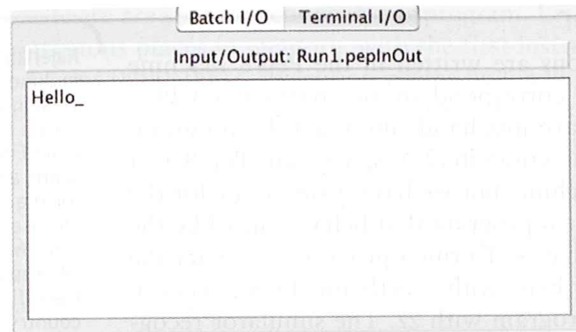


Click on the middle of these three icons, which calls the loader. After you click on this icon, your program is loaded into the Pep/8 memory.

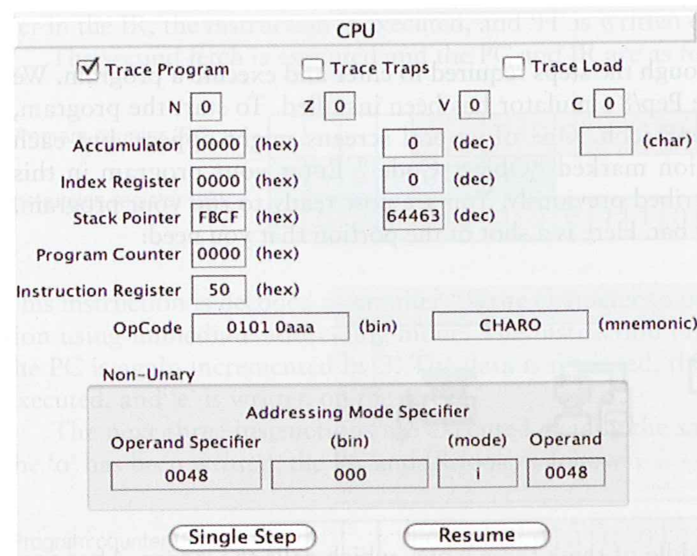
Be sure the Terminal I/O button is darkened (pressed). Now click on the rightmost icon, which is the execute button. The program is executed and "Hello" appears in the output window. For everything that we do in this chapter, the Terminal I/O button should be darkened. This area is where you input and output values.

? **Nigerian check scams**
In June 2008, Edna Fiedler of Olympia, Washington, was sentenced to 2 years in prison and 5 years of supervised probation in a $1 million Nigerian check scam. In this scam, a message in broken English pleaded for the financial help of the kind recipient. In all cases, the sender was a high-ranking official who had millions stashed in an inaccessible spot. If the recipient wired money for the official's escape from his ravaged country, he or she was offered a share of the money. The average loss to the victims of this scam was more than $5000.

Loader A piece of software that takes a machine-language program and places it into memory



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.

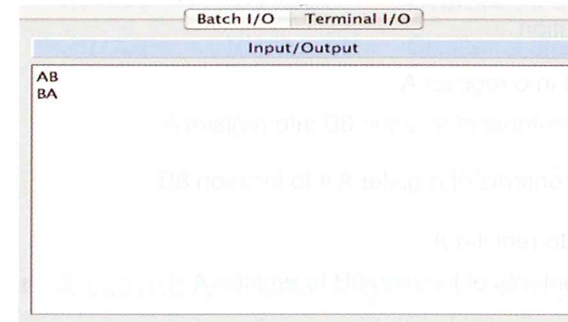


When the "Trace Program" option is checked, press the Single Step button and the first instruction will be executed. Continue pressing the Single Step button, and you can see the register values change.

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