7.28 ***(A Computer Simulator)*** It may at first seem outrageous, but in this problem you're going to build your own computer. No, you won't be soldering components together. Rather, you'll use the powerful technique of *software-based simulation* to create a *software model* of the Simpletron. You'll not be disappointed. Your Simpletron simulator will turn the computer you're using into a Simpletron, and you'll actually be able to run, test and debug the SML programs you wrote in Exercise 7.27.

When you run your Simpletron simulator, it should begin by printing:

```
*** Welcome to Simpletron! ***
*** Please enter your program one instruction ***
*** (or data word) at a time. I will type the ***
*** location number and a question mark (?).  ***
*** You then type the word for that location. ***
*** Type the sentinel -99999 to stop entering ***
*** your program. ***
```

Simulate the memory of the Simpletron with a single-subscripted array `memory` that has 100 elements. Now assume that the simulator is running, and let's examine the dialog as we enter the program of Example 2 of Exercise 7.27:

```
00 ? +1009
01 ? +1010
02 ? +2009
03 ? +3110
04 ? +4107
05 ? +1109
06 ? +4300
07 ? +1110
08 ? +4300
09 ? +0000
10 ? +0000
11 ? -99999
*** Program loading completed ***
*** Program execution begins  ***
```

The SML program has now been placed (or loaded) into the array `memory`. Now the Simpletron executes the SML program. It begins with the instruction in location `00` and continues sequentially, unless directed to some other part of the program by a transfer of control.

Use the variable `accumulator` to represent the accumulator register. Use the variable `instructionCounter` to keep track of the location in memory that contains the instruction being performed. Use the variable `operationCode` to indicate the operation currently being performed—i.e., the left two digits of the instruction word. Use the variable `operand` to indicate the memory location on which the current instruction operates. Thus, if an instrucion has an `operand`, it's the rightmost two digits of the instruction currently being performed. Do not execute instructions directly from memory. Rather, transfer the next instruction to be performed from memory to a variable called `instructionRegister`. Then "pick off" the left two digits and place them in the variable `operationCode`, and "pick off " the right two digits and place them in `operand`.

When Simpletron begins execution, the special registers are initialized as follows:

```
accumulator                +0000
instructionCounter            00
instructionRegister        +0000
operationCode                 00
operand                       00
```

Now let's "walk through" the execution of the first SML instruction, `+1009` in memory location `00`. This is called an *instruction execution cycle.*

The `instructionCounter` tells us the location of the next instruction to be performed. We *fetch* the contents of that location from `memory` by using the C statement

```
instructionRegister = memory[ instructionCounter ];
```

The operation code and the operand are extracted from the instruction register by the statements

```
operationCode = instructionRegister / 100;
operand = instructionRegister % 100;
```

Now the Simpletron must determine that the operation code is actually a *read* (versus a *write*, a *load*, and so on). A `switch` differentiates among the twelve operations of SML.

The `switch` statement simulates the behavior of various SML instructions as follows (we leave the others to the reader):

read: `scanf( "%d", &memory[ operand ] );`
load: `accumulator = memory[ operand ];`
add: `accumulator += memory[ operand ];`
Various branch instructions: We'll discuss these shortly.
halt: This instruction prints the message

```
*** Simpletron execution terminated ***
```

then prints the name and contents of each register as well as the complete contents of memory. Such a printout is often called a *computer dump*. To help you program your dump function, a sample dump format is shown in Fig. 7.33. A dump after executing a Simpletron program would show the actual values of instructions and data values at the moment execution terminated. You can print leading 0s in front of an integer that is shorter than its field width by placing the 0 formatting flag before the field width in the format specifier as in `"%02d"`. You can place a + or - sign before a value with the + formatting flag. So to produce a number of the form +0000, you can use the format specifier `"%+05d"`.

Let's proceed with the execution of our program's first instruction, namely the `+1009` in location `00`. As we've indicated, the `switch` statement simulates this by performing the C statement

```
scanf( "%d", &memory[ operand ] );
```

A question mark (?) should be displayed on the screen before the `scanf` is executed to prompt the user for input. The Simpletron waits for the user to type a value and then press the *Return* key. The value is then read into location `09`.

At this point, simulation of the first instruction is completed. All that remains is to prepare the Simpletron to execute the next instruction. Because the instruction just performed was not a transfer of control, we need merely increment the instruction counter register as follows:

```
++instructionCounter;
```

This completes the simulated execution of the first instruction. The entire process (i.e., the instruction execution cycle) begins anew with the fetch of the next instruction to be executed.

Now let's consider how the branching instructions—the transfers of control—are simulated. All we need to do is adjust the value in the instruction counter appropriately. Therefore, the unconditional branch instruction (`40`) is simulated within the `switch` as

```
instructionCounter = operand;
```

The conditional "branch if accumulator is zero" instruction is simulated as

```
if ( accumulator == 0 ) {
   instructionCounter = operand;
}
```

At this point, you should implement your Simpletron simulator and run the SML programs you wrote in Exercise 7.27. You may embellish SML with additional features and provide for these in your simulator.

```
REGISTERS:
accumulator              +0000
instructionCounter          00
instructionRegister      +0000
operationCode               00
operand                     00

MEMORY:
         0      1      2      3      4      5      6      7      8      9
 0   +0000  +0000  +0000  +0000  +0000  +0000  +0000  +0000  +0000  +0000
10   +0000  +0000  +0000  +0000  +0000  +0000  +0000  +0000  +0000  +0000
20   +0000  +0000  +0000  +0000  +0000  +0000  +0000  +0000  +0000  +0000
30   +0000  +0000  +0000  +0000  +0000  +0000  +0000  +0000  +0000  +0000
40   +0000  +0000  +0000  +0000  +0000  +0000  +0000  +0000  +0000  +0000
50   +0000  +0000  +0000  +0000  +0000  +0000  +0000  +0000  +0000  +0000
60   +0000  +0000  +0000  +0000  +0000  +0000  +0000  +0000  +0000  +0000
70   +0000  +0000  +0000  +0000  +0000  +0000  +0000  +0000  +0000  +0000
80   +0000  +0000  +0000  +0000  +0000  +0000  +0000  +0000  +0000  +0000
90   +0000  +0000  +0000  +0000  +0000  +0000  +0000  +0000  +0000  +0000
```

Fig. 7.33 | Sample Simpletron dump format.

Your simulator should check for various types of errors. During the program loading phase, for example, each number the user types into the Simpletron's `memory` must be in the range `-9999` to `+9999`. Your simulator should use a `while` loop to test that each number entered is in this range, and, if not, keep prompting the user to reenter the number until a correct number is entered.

During the execution phase, your simulator should check for serious errors, such as attempts to divide by zero, attempts to execute invalid operation codes and accumulator overflows (i.e., arithmetic operations resulting in values larger than `+9999` or smaller than `-9999`). Such serious errors are called *fatal errors*. When a fatal error is detected, print an error message such as:

```
*** Attempt to divide by zero ***
*** Simpletron execution abnormally terminated ***
```

and should print a full computer dump in the format we've discussed previously. This will help the user locate the error in the program.

Implementation Note: When you implement the Simpletron Simulator, define the `memory` array and all the registers as variables in `main`. The program should contain three other functions—`load`, `execute` and `dump`. Function `load` reads the SML instructions from the user at the keyboard. (Once you study file processing in Chapter 11, you'll be able to read the SML instruction from a file.) Function `execute` executes the SML program currently loaded in the `memory` array. Function `dump` displays the contents of `memory` and all of the registers stored in `main`'s variables. Pass the `memory` array and registers to the other functions as necessary to complete their tasks. Functions `load` and `execute` need to modify variables that are defined in `main`, so you'll need to pass those variables to the functions by reference using pointers. So, you'll need to modify the statements we showed throughout this problem description to use the appropriate pointer notation.