In the next several problems, we take a temporary diversion away from the world of high-level-language programming. We "peel open" a computer and look at its internal structure. We introduce machine-language programming and write several machine-language programs. To make this an especially valuable experience, we then build a computer (through the technique of software-based *simulation)* on which you can execute your machine-language programs!

5.18 (*Machine-Language Programming*) Let us create a computer we will call the Simpletron. As its name implies, it is a simple machine, but, as we will soon see, a powerful one as well. The Simpletron runs programs written in the only language it directly understands; that is, Simpletron Machine Language, or SML for short.

The Simpletron contains an *accumulator*—a "special register" in which information is put before the Simpletron uses that information in calculations or examines it in various ways. All information in the Simpletron is handled in terms of *words.* A word is a signed four-digit decimal number, such as `+3364`, `-1293`, `+0007`, `-0001`, etc. The Simpletron is equipped with a 100-word memory, and these words are referenced by their location numbers `00`, `01`, …, `99`.

Before running an SML program, we must *load,* or place, the program into memory. The first instruction (or statement) of every SML program is always placed in location `00`. The simulator will start executing at this location.

Each instruction written in SML occupies one word of the Simpletron's memory. (Thus, instructions are signed four-digit decimal numbers.) We shall assume that the sign of an SML instruction is always plus, but the sign of a data word may be either plus or minus. Each location in the Simpletron's memory may contain an instruction, a data value used by a program or an unused (and hence undefined) area of memory. The first two digits of each SML instruction are the *operation code* that specifies the operation to be performed. SML operation codes are shown in Fig. 5.37.

Operation code	Meaning
Input/output operations:	
`const int READ = 10`	Read a word from the keyboard into a specific location in memory.
`const int WRITE = 11;`	Write a word from a specific location in memory to the screen.
Load and store operations:	
`const int LOAD = 20;`	Load a word from a specific location in memory into the accumulator.
`const int STORE = 21;`	Store a word from the accumulator into a specific location in memory.
Arithmetic operations:	
`const int ADD = 30;`	Add a word from a specific location in memory to the word in the accumulator (leave result in accumulator).
`const int SUBTRACT = 31;`	Subtract a word from a specific location in memory from the word in the accumulator (leave result in accumulator).
`const int DIVIDE = 32;`	Divide a word from a specific location in memory into the word in the accumulator (leave result in accumulator).
`const int MULTIPLY = 33;`	Multiply a word from a specific location in memory by the word in the accumulator (leave result in accumulator).
Transfer of control operations:	
`const int BRANCH = 40;`	Branch to a specific location in memory.

Fig. 5.37 Simpletron Machine Language (SML) operation codes (part 1 of 2).

Operation code	Meaning
`const int BRANCHNEG = 41;`	Branch to a specific location in memory if the accumulator is negative.
`const int BRANCHZERO = 42;`	Branch to a specific location in memory if the accumulator is zero.
`const int HALT = 43;`	Halt—the program has completed its task.

Fig. 5.37 Simpletron Machine Language (SML) operation codes (part 2 of 2).

The last two digits of an SML instruction are the *operand*—the address of the memory location containing the word to which the operation applies.

Now let us consider several simple SML programs. The first SML program (Example 1) reads two numbers from the keyboard and computes and prints their sum. The instruction **`+1007`** reads the first number from the keyboard and places it into location **`07`** (which has been initialized to zero). Then instruction **`+1008`** reads the next number into location **`08`**. The *load* instruction, **`+2007`**, puts (copies) the first number into the accumulator, and the *add* instruction, **`+3008`**, adds the second number to the number in the accumulator. *All SML arithmetic instructions leave their results in the accumulator*. The *store* instruction, **`+2109`**, places (copies) the result back into memory location **`09`** from which the *write* instruction, **`+1109`**, takes the number and prints it (as a signed four-digit decimal number). The *halt* instruction, **`+4300`**, terminates execution.

Example 1 Location	Number	Instruction
00	+1007	(Read A)
01	+1008	(Read B)
02	+2007	(Load A)
03	+3008	(Add B)
04	+2109	(Store C)
05	+1109	(Write C)
06	+4300	(Halt)
07	+0000	(Variable A)
08	+0000	(Variable B)
09	+0000	(Result C)

The SML program in Example 2 reads two numbers from the keyboard and determines and prints the larger value. Note the use of the instruction **`+4107`** as a conditional transfer of control, much the same as C++'s **`if`** statement.

Example 2 Location	Number	Instruction
00	+1009	(Read A)
01	+1010	(Read B)
02	+2009	(Load A)
03	+3110	(Subtract B)
04	+4107	(Branch negative to 07)
05	+1109	(Write A)
06	+4300	(Halt)
07	+1110	(Write B)

Example 2 Location	Number	Instruction
`08`	`+4300`	`(Halt)`
`09`	`+0000`	`(Variable A)`
`10`	`+0000`	`(Variable B)`

Now write SML programs to accomplish each of the following tasks.

a) Use a sentinel-controlled loop to read positive numbers and compute and print their sum. Terminate input when a negative number is entered.

b) Use a counter-controlled loop to read seven numbers, some positive and some negative, and compute and print their average.

c) Read a series of numbers and determine and print the largest number. The first number read indicates how many numbers should be processed.

5.19 (*A Computer Simulator*) It may at first seem outrageous, but in this problem, you are going to build your own computer. No, you will not be soldering components together. Rather, you will use the powerful technique of *software-based simulation* to create a *software model* of the Simpletron. You will not be disappointed. Your Simpletron simulator will turn the computer you are using into a Simpletron, and you will actually be able to run, test and debug the SML programs you wrote in Exercise 5.18.

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 us examine the dialog as we enter the program of Example 2 of Exercise 5.18:

```
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) in array **`memory`**. Now the Simpletron executes your SML program. Execution begins with the instruction in location **`00`** and, like C++, 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 **`counter`** to keep track of the location in memory that contains the instruction being performed. Use variable **`operationCode`** to indicate the operation currently being performed (i.e., the left two digits of the instruction word). Use variable **`operand`** to indicate the memory location on which the current instruction operates. Thus, **`operand`** is 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 **`operationCode`**, and "pick off" the right two digits and place them in **`operand`**. When Simpletron begins execution, the special registers are all initialized to zero.

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

The **counter** 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[ counter ];
```

The operation code and 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*, etc.). A **switch** differentiates among the twelve operations of SML.

In the **switch** structure, the behavior of various SML instructions is simulated as follows (we leave the others to the reader):

read: **cin >> memory[operand];**

load: **accumulator = memory[operand];**

add: **accumulator += memory[operand];**

branch: We will discuss the branch instructions shortly.

halt: This instruction prints the message
***** Simpletron execution terminated *****

It 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* (and, no, a computer dump is not a place where old computers go). To help you program your dump function, a sample dump format is shown in Fig. 5.38. Note that a dump after executing a Simpletron program would show the actual values of instructions and data values at the moment execution terminated.

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

```
cin >> memory[ operand ];
```

A question mark (**?**) should be displayed on the screen before the **cin** 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. Since the instruction just performed was not a transfer of control, we need merely increment the instruction counter register as follows:

```
++counter;
```

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 us 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

```
counter = operand;
```

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

```
if ( accumulator == 0 )
   counter = operand;
```

At this point you should implement your Simpletron simulator and run each of the SML programs you wrote in Exercise 5.18. You may embellish SML with additional features and provide for these in your simulator.

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 the user enters a correct number.

```
REGISTERS:
accumulator          +0000
counter                 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. 5.38 A sample dump.

During the execution phase, your simulator should check for various serious errors, such as attempts to divide by zero, attempts to execute invalid operation codes, accumulator overflows (i.e., arithmetic operations resulting in values larger than **+9999** or smaller than **-9999**) and the like. Such serious errors are called *fatal errors.* When a fatal error is detected, your simulator should 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 have discussed previously. This will help the user locate the error in the program.

```
// Exercise 5.19 Solution
#include <iostream>

using std::cout;
using std::endl;
using std::cin;
using std::ios;

#include <iomanip>

using std::setfill;
using std::setw;
using std::setiosflags;
using std::resetiosflags;

const int SIZE = 100, MAX_WORD = 9999, MIN_WORD = -9999;
const long SENTINEL = -99999;
enum Commands { READ = 10, WRITE, LOAD = 20, STORE, ADD = 30, SUBTRACT,
                DIVIDE, MULTIPLY, BRANCH = 40, BRANCHNEG, BRANCHZERO, HALT };

void load( int * const );
void execute( int * const, int * const, int * const, int * const,
```

```
               int * const, int * const);
void dump( const int * const, int, int, int, int, int );
bool validWord( int );

int main()
{
   int memory[ SIZE ] = { 0 }, accumulator = 0, instructionCounter = 0,
       opCode = 0, operand = 0, instructionRegister = 0;

   load( memory );
   execute( memory, &accumulator, &instructionCounter, &instructionRegister,
            &opCode, &operand );
   dump( memory, accumulator, instructionCounter, instructionRegister,
        opCode, operand );

   return 0;
}

void load( int * const loadMemory )
{
   long instruction;
   int i = 0;

   cout << "***           Welcome to Simpletron           ***\n"
        << "*** Please enter your program one instruction ***\n"
        << "*** (or data word) at a time. I will type the ***\n"
        << "*** location number and a question mark (?).  ***\n"
        << "*** You then type the word for that location. ***\n"
        << "*** Type the sentinel -99999 to stop entering ***\n"
        << "*** your program.                             ***\n" << "00 ? ";
   cin >> instruction;

   while ( instruction != SENTINEL ) {

      if ( !validWord( instruction ) )
         cout << "Number out of range. Please enter again.\n";
      else
         loadMemory[ i++ ] = instruction;

      // function setfill sets the padding character for unused
      // field widths.
      cout << setw( 2 ) << setfill( '0' ) << i << " ? ";
      cin >> instruction;
   }
}

void execute( int * const memory, int * const acPtr, int * const icPtr,
              int * const irPtr, int * const opCodePtr, int * const opPtr )
{
   bool fatal = false;
   int temp;
   const char *messages[] = { "Accumulator overflow          ***",
                              "Attempt to divide by zero     ***",
                              "Invalid opcode detected       ***" },
         *termString = "\n*** Simpletron execution abnormally terminated ***",
         *fatalString = "*** FATAL ERROR: ";

   cout << "\n************START SIMPLETRON EXECUTION************\n\n";

   do {
      *irPtr = memory[ *icPtr ];
      *opCodePtr = *irPtr / 100;
```

```
      *opPtr = *irPtr % 100;

      switch ( *opCodePtr ) {
         case READ:
            cout << "Enter an integer: ";
            cin >> temp;

            while ( !validWord( temp ) ) {
               cout << "Number out of range. Please enter again: ";
               cin >> temp;
            }

            memory[ *opPtr ] = temp;
            ++( *icPtr );
            break;
         case WRITE:
            cout << "Contents of " << setw( 2 ) << setfill( '0' ) << *opPtr
                 << ": " << memory[ *opPtr ] << '\n';
            ++( *icPtr );
            break;
         case LOAD:
            *acPtr = memory[ *opPtr ];
            ++( *icPtr );
            break;
         case STORE:
            memory[ *opPtr ] = *acPtr;
            ++( *icPtr );
            break;
         case ADD:
            temp = *acPtr + memory[ *opPtr ];

            if ( !validWord( temp ) ) {
               cout << fatalString << messages[ 0 ] << termString << '\n';
               fatal = true;
            }
            else {
               *acPtr = temp;
               ++( *icPtr );
            }

            break;
         case SUBTRACT:
            temp = *acPtr - memory[ *opPtr ];

            if ( !validWord( temp ) ) {
               cout << fatalString << messages[ 0 ] << termString << '\n';
               fatal = true;
            }
            else {
               *acPtr = temp;
               ++( *icPtr );
            }

            break;
         case DIVIDE:
            if ( memory[ *opPtr ] == 0 ) {
               cout << fatalString << messages[ 1 ] << termString << '\n';
               fatal = true;
            }
            else {
               *acPtr /= memory[ *opPtr ];
               ++( *icPtr );
            }
```

```cpp

            break;
         case MULTIPLY:
            temp = *acPtr * memory[ *opPtr ];

            if ( !validWord( temp ) ) {
               cout << fatalString << messages[ 0 ] << termString << '\n';
               fatal = true;
            }
            else {
               *acPtr = temp;
               ++( *icPtr );
            }
            break;
         case BRANCH:
            *icPtr = *opPtr;
            break;
         case BRANCHNEG:
            *acPtr < 0 ? *icPtr = *opPtr : ++( *icPtr );
            break;
         case BRANCHZERO:
            *acPtr == 0 ? *icPtr = *opPtr : ++( *icPtr );
            break;
         case HALT:
            cout << "*** Simpletron execution terminated ***\n";
            break;
         default:
            cout << fatalString << messages[ 2 ] << termString << '\n';
            fatal = true;
            break;
      }
   } while ( *opCodePtr != HALT && !fatal );

   cout << "\n*************END SIMPLETRON EXECUTION*************\n";
}

void dump( const int * const memory, int accumulator, int instructionCounter,
          int instructionRegister, int operationCode, int operand )
{
   void output( const char * const, int, int, bool );   // prototype

   cout << "\nREGISTERS:\n";
   output( "accumulator", 5, accumulator, true );
   output( "instructionCounter", 2, instructionCounter, false );
   output( "instructionRegister", 5, instructionRegister, true );
   output( "operationCode", 2, operationCode, false );
   output( "operand", 2, operand, false );
   cout << "\n\nMEMORY:\n";

   int i = 0;
   cout << setfill( ' ' ) << setw( 3 ) << ' ';

   // print header
   for ( ; i <= 9; ++i )
      cout << setw( 5 ) << i << ' ';

   for ( i = 0; i < SIZE; ++i ) {
      if ( i % 10 == 0 )
         cout << '\n' << setw( 2 ) << i << ' ';

      cout << setiosflags( ios::internal | ios::showpos )
           << setw( 5 ) << setfill( '0' ) << memory[ i ] << ' '
           << resetiosflags( ios::internal | ios::showpos );
```

```
   }

   cout << endl;
}

bool validWord( int word )
{
   return word >= MIN_WORD && word <= MAX_WORD;
}

void output( const char * const sPtr, int width, int value, bool sign )
{
   // format of "accumulator", etc.
   cout << setfill( ' ' ) << setiosflags( ios::left ) << setw( 20 )
        << sPtr << ' ';

   // is a +/- sign needed?
   if ( sign )
      cout << setiosflags( ios::showpos | ios::internal );

   // setup for displaying accumulator value, etc.
   cout << resetiosflags( ios::left ) << setfill( '0' );

   // determine the field widths and display value
   if ( width == 5 )
      cout << setw( width ) << value << '\n';
   else  // width is 2
      cout << setfill( ' ' ) << setw( 3 ) << ' ' << setw( width )
           << setfill( '0' ) << value << '\n';

   // disable sign if it was set
   if ( sign )
      cout << resetiosflags( ios::showpos | ios::internal );
}
```

```
***           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.                             ***
00 ? 1099
01 ? 1098
02 ? 2099
03 ? 3398
04 ? 2150
05 ? 1150
06 ? 1199
07 ? 1198
08 ? 4300
09 ? -99999

************START SIMPLETRON EXECUTION************

Enter an integer: 4
Enter an integer: 9
Contents of 50: 36
Contents of 99: 4
Contents of 98: 9
*** Simpletron execution terminated ***

*************END SIMPLETRON EXECUTION*************

REGISTERS:
accumulator          +0036
instructionCounter      08
instructionRegister  +4300
operationCode           43
operand                 00


MEMORY:
       0     1     2     3     4     5     6     7     8     9
 0 +1099 +1098 +2099 +3398 +2150 +1150 +1199 +1198 +4300 +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 +0036 +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 +0009 +0004
```