Assignment 4

This assignment is the first in a sequence of three. It is not strictly necessary to complete this one in order to do the other two, but the understanding you gain in completing this assignment will make writing the third assignment (a major project) much easier. You will need to complete the second in the sequence in order to do the third.

We start by "peeling open" a computer, look at its internal structure, and introducing machine language (assembler-level) programming. Your assignment is to write a program that simulates a computer, one that is capable of executing machine language programs.

1. Initial Setup

1. Log in to Unix.
2. Run the `setup` script for Assignment 4 by typing:

   ```
   setup 4
   ```

2. Description of the Simplesim Computer

In this assignment you will write a program to simulate a fictional computer that we will call the Simplesim. As its name implies it is a simple machine. All information in the Simplesim is handled in terms of words. A word is a signed four-digit decimal (base 10) number such as +3364, -1293, +0007, -0001, 0000, etc. The Simplesim is equipped with memory and five registers.

- The Simplesim has a 100-word memory and these words are referenced by their location numbers 00, 01, . . . , 99. Each word in the Simplesim's memory (always a single signed four-digit decimal number) may be interpreted as an instruction to be executed, a data value, or may be uninitialized.
- The first register is the *accumulator*, which is just large enough to hold a single word. Words from memory must be placed into the accumulator in order to perform arithmetic on them or test their values. All arithmetic and branching is done using the accumulator.
- The second register is the *instruction counter*, which is just large enough to hold a memory location (a two digit number, 00, 01, ... , 99). The instruction counter is used to hold the memory location of the next instruction to be executed.
- The third register is the *instruction register*, which, like the accumulator, is just large enough to hold a single word. The instruction register is used to hold a copy of the instruction (a word that was pulled out of memory) that is currently being executed.

- The fourth and fifth registers are the *operation code* and *operand*, respectively. Each one is just large enough to hold half of a word (a two digit decimal number). The operation code and operand registers are used to "split" the instruction register in half, with the 2 leftmost digits and sign of the instruction register going into the operation code and the 2 rightmost digits going into the operand. For example, if the instruction register had +1009, the operation code would have +10 and the operand would have 09. Likewise, if the instruction register had -1201, the operation code would have -12 and the operand would have 01.

3. The Simplesim Machine Language (SML)

Each instruction written in the Simplesim Machine Language (SML) occupies one word of the Simplesim's memory (and hence instructions are signed four-digit decimal numbers). The two leftmost digits of each SML instruction are the *operation code* (opcode), which specifies the operation to be performed. The two rightmost digits of an SML instruction are the *operand*, which is the memory location containing the word to which the operation applies. The complete set of SML instructions is described in the table that follows.

Operation Code	Meaning
Input / Output Operations:	
`#define READ 01`	Read a word into a specific memory location.
`#define WRITE 02`	Print a word from a specific memory location.
Store / Load Operations:	
`#define STORE 11`	Store the word in the accumulator into a specific memory location.
`#define LOAD 12`	Load a word from a specific memory location into the accumulator.
Arithmetic Operations:	
`#define ADD 21`	Add a word in a specific memory location to the word in the accumulator (leave result in accumulator).
`#define SUBTRACT 22`	Subtract a word in a specific memory location from the word in the accumulator (leave result in accumulator).
`#define MULTIPLY 23`	Multiply a word in a specific memory location by the word in the accumulator (leave result in accumulator).
`#define DIVIDE 24`	Divide a word in a specific memory location into the word in the accumulator (leave result in accumulator).
Transfer of Control Operations:	
`#define BRANCH 31`	Branch to a specific memory location.
`#define BRANCHZERO 32`	Branch to a specific memory location if the accumulator is

	zero.
`#define BRANCHNEG 33`	Branch to a specific memory location if the accumulator is negative.
`#define HALT 34`	Halt, i.e., the program has completed its task.

We illustrate how the Simplesim executes SML programs (using the instructions from the table above) with the use of two example SML programs. Consider the following SML program which reads two numbers and computes and prints their sum.

Memory Location	**Word**	**Instruction**
00	+0107	(Read A)
01	+0108	(Read B)
02	+1207	(Load A)
03	+2108	(Add B)
04	+1109	(Store C)
05	+0209	(Write C)
06	+3400	(Halt)
07	+0000	(Variable A)
08	+0000	(Variable B)
09	+0000	(Result C)

Execution always begins at memory location 00. The word at memory location 00 (+0107) is read and interpreted as an instruction. The leftmost two digits of the word (01) represent the instruction and the rightmost two digits (07) represent the instruction's operand. The first instruction is a `READ` operation. This reads a single word from the input file (explained in **Section 4**) and stores it in the memory location defined by the operand, in this case memory location 07. `READ` and `WRITE` instructions always operate on memory locations. This completes the execution of the first instruction. Processing continues by executing the next instruction found at memory location 01.

The next instruction (+0108) reads a second word from the input file and stores it in memory location 08. The next instruction (+1207) is a `LOAD` operation with operand 07. It takes the word found at memory location 07 (the operand) and places it into the accumulator (recall that the accumulator is one of the five registers described in **Section 1**). All `LOAD` and `STORE` operations move data in and out of the accumulator.

The next instruction (+2108) is an `ADD` instruction with operand 08. All SML arithmetic instructions are performed using the word in the accumulator and the word identified by the operand and the result is always left in the accumulator. This instruction takes the word stored in memory location 08 (the operand), adds it to the value in the accumulator, and leaves the sum in the accumulator.

The next instruction (+1109) is a `STORE` instruction which, like all `STORE` instructions, takes the word in the accumulator (the sum of the two input values) and stores it in the memory location identified by the instruction's operand, in this case memory location 09. Then +0209, a `WRITE` instruction, prints (output is explained in **Section 5**) the word found in memory location 09, which - again - is the sum of the two input values. Finally instruction +3400, the `HALT` instruction, is executed which simply terminates the SML program (operand 00 is ignored for this instruction).

Note that a single word in memory can be used to store a single instruction that is to be executed or a single variable that should never be interpreted as an instruction. None of the memory locations after the `HALT` instruction (memory locations 07-09) were executed; however, they were important in the computation. Those words were used to store the program's variables and temporary results.

All SML programs will "partition" the Simplesim's memory in this way. The first words of memory (always starting at memory location 00) are the "instructions" of the program and following that, after the `HALT` instruction, is the "data" part of the program. The intention, of course, is that only the "instructions" of the program are to be executed, i.e., each word interpreted as an SML instruction.

Now consider this second SML program that reads two numbers and prints the larger of the two.

Memory Location	Word	Instruction
00	+0109	(Read A)
01	+0110	(Read B)
02	+1209	(Load A)
03	+2210	(Subtract B)
04	+3207	(Branch negative to 07)
05	+0209	(Write A)
06	+3400	(Halt)
07	+0210	(Write B)
08	+3400	(Halt)
09	+0000	(Variable A)
10	+0000	(Variable B)

The first two instructions (+0109 and +0110) read two values and store them in memory locations 09 and 10, respectively. +1209 places the word at memory location 09 (the first input value) into the accumulator. +2210, a `SUBTRACT` instruction, takes the word at memory location 10 (the second input value), subtracts it from the accumulator, and leaves the result in the accumulator.

+3207 (`BRANCHNEG`) is a conditional branch instruction, much like an "if" statement in C++. All conditional branch instructions are based on the accumulator. The `BRANCH` instruction, which acts like a "goto", is the only branch instruction that ignores the accumulator; it is simply an unconditional branch.

If the value in the accumulator is negative, which in this case means the second input value was the largest, then the next instruction that gets executed is the one at memory location 07 (the operand). If the value in the accumulator is 0 or greater, meaning the first input value was greater than or equal to the second, then execution continues with the next statement, i.e., no branching. If the branch was taken, then the value at memory location 10 (the second input value) is printed and the program terminates. Otherwise the value at memory location 09 (the first input value) is printed and the program terminates.

Note how the SML program is written. It "partitions" the Simplesim's memory into two distinct parts; the "program" (locations 00-08) and the "data" (locations 09-10). This SML program, unlike the first, has two `HALT` instructions. This is okay; only one of them will be executed. The point is that `HALT` instructions are used to prevent the execution of the program from wandering into the "data" portion of the program.

4. Input

Your program will take as input an SML program followed by any input for that SML program.

The input file will start with the SML program, one instruction per line. Following the last line of the SML program will be the number -99999, which is not part of the SML program. If the SML program expects any input (i.e., if it has any `READ` instructions) then input for the SML program, one input value per line, immediately follows the -99999 line. For example, below is the input file for the first program from the previous section. It adds -5 and 15.

```
0107
0108
1207
2108
1109
0209
3400
0000
0000
0000
-99999
-5
15
```

Note that each line of the input file, other than -99999 which is used to denote the end of program and not intended to be placed into the Simplesim's memory, fits into a single word. Note also that not all SML programs require input (those that do not have `READ` instructions). In that case there would be no data after the -99999 line.

All input files to your program have -99999 after the last SML instruction. For those SML programs that do not require input, (those that do not have `READ` instructions), -99999 is simply the last line of the input file.

5. Output

Each time a `READ` instruction is executed your program must print the value that was read. For example, the two values read in the program from the previous section are -5 and 15. As each value is read, your program should print output that looks exactly like this.

```
READ: -0005
READ: +0015
```

For each `WRITE` instruction your program must print the value of the word in that memory location. For example, from the program in the previous section, the sum 10 is printed exactly like this.

```
+0010
```

When the `HALT` statement is executed, your program should print the following line:

```
*** Simplesim execution terminated ***
```

At the end of any execution your program should dump the entire contents of the Simplesim. This means dumping the contents of all five registers and all 100 words of the Simplesim's memory.

Assuming that the name of your program is `simplesim` and the name of the SML program file above is `sum.sml`, then the output of your program must look exactly like this:

```
z123456@turing:~/csci241/Assign4$ ./simplesim < sum.sml
READ: -0005
READ: +0015
+0010
*** Simplesim execution terminated ***

REGISTERS:
accumulator:            +0010
instruction_counter:    06
instruction_register:   +3400
operation_code:         34
operand:                00

MEMORY:
       0     1     2     3     4     5     6     7     8     9
 0 +0107 +0108 +1207 +2108 +1109 +0209 +3400 -0005 +0015 +0010
10 +4444 +4444 +4444 +4444 +4444 +4444 +4444 +4444 +4444 +4444
20 +4444 +4444 +4444 +4444 +4444 +4444 +4444 +4444 +4444 +4444
30 +4444 +4444 +4444 +4444 +4444 +4444 +4444 +4444 +4444 +4444
40 +4444 +4444 +4444 +4444 +4444 +4444 +4444 +4444 +4444 +4444
50 +4444 +4444 +4444 +4444 +4444 +4444 +4444 +4444 +4444 +4444
60 +4444 +4444 +4444 +4444 +4444 +4444 +4444 +4444 +4444 +4444
70 +4444 +4444 +4444 +4444 +4444 +4444 +4444 +4444 +4444 +4444
80 +4444 +4444 +4444 +4444 +4444 +4444 +4444 +4444 +4444 +4444
```

```
90 +4444 +4444 +4444 +4444 +4444 +4444 +4444 +4444 +4444 +4444
z123456@turing:~/csci241/Assign4$
```

One of the first things that your program will do is read the SML program into the Simplesim's memory. This is called *loading* the program. There are a couple of things that could go wrong when loading the program; the program may be too large for the Simplesim's 100-word memory or a line of the input file may not fit into a word (i.e., it may be greater than 9999 or less than -9999). In these situations your program should print an error message, dump the contents of the machine, and terminate. It should not start to run the SML program.

If there was a successful SML program load, your program should start to execute the SML program. SML programs, like any other programs, may perform an illegal operation and terminate abnormally (abend). There are a number of conditions that may cause an SML program to abend, in which case processing stops immediately An example of this is an attempt to divide by 0. In that case, the Simplesim should print an appropriate abend message, stop execution, and dump the contents of the machine. Every execution of your program (normal termination of the SML program or SML program abend) ends with a dump of the Simplesim.

A summary of the possible abend conditions (program load and execution errors) with their error messages appear in the following table. Note that all error messages must appear exactly as they appear in the table.

Condition	Error Message	Description
Program Load Errors:		
Program too big	`*** ABEND: pgm load: pgm too large ***`	The program is too big (more than 100 words) to fit into memory.
Invalid word	`*** ABEND: pgm load: invalid word ***`	During program load, one of the words in the input file was less than -9999 or greater than 9999.
Execution Errors:		
Invalid opcode	`*** ABEND: invalid opcode ***`	An attempt was made to execute an unrecognizable instruction, i.e., the leftmost two digits of the word was not a valid instruction.
Adressability	`*** ABEND: addressability error ***`	An attempt was

		made to fetch an instruction from an invalid memory location.
Division by 0	`*** ABEND: attempted division by 0 ***`	Attempt to divide by 0.
Underflow	`*** ABEND: underflow ***`	The result of an arithmetic operation is less than -9999, and therefore would not fit into the accumulator.
Overflow	`*** ABEND: overflow ***`	The result of an arithmetic operation is greater than 9999, and therefore would not fit into the accumulator.
Illegal input	`*** ABEND: illegal input ***`	During a `READ` instruction an attempt was made to read a value that was either less than -9999 or greater than 9999.

```
#ifndef SML_H
#define SML_H
//sml.h
//assignment 3

//created by vanessa

#define READ 10
#define write 11

etc.


#endif
saml.h //save as
```

```
#ifndef SML_H
#define SML_H
//sml.h
//assignment 3

//created by vanessa


class simplesim
{

public:

simplesim():
void execute_program();
bool load_program();
void dump_program() const;

private:

int memeory[100];
int accumulator =0;
int instruction_counter=0;
int instruction_register=0;
int operation_code=0;
int operand=0;

#endif


simplesimh //save as
```

6. Files You Must Write

You will need to write three files for this assignment:

- `sml.h` - This header file must contain all the `#define` statements that define the Simplesim's instruction set. i.e., `#define READ 10`, `#define WRITE 11`, etc. The header file should have an appropriate set of header guards to prevent it from being included multiple times in the same source file.

- `simplesim.h` - This header file should contain the class definition for a class called `aimplesim`. This class definition should contain the private data members described below under **7. Simulating the Simplesim**. It should also contain public declarations (i.e., prototypes) for the four member functions whose definitions are contained in `simplesim.cpp` (described below). The header file should have an appropriate set of header guards.

- `simplesim.cpp` - This source file will contain function definitions for the following four member functions of the `simplesim` class:

    - `simplesim::simplesim()`

      This "default constructor" takes no arguments and has no return value. Its job is to perform the initialization process for a `simplesim` object described under **7.1. Initialize Simplesim**.

    - `bool simplesim::load_program()`

      This member function reads an SML program from standard input and attempts to load it into the Simplesim's memory, as described under **7.2. Load SML Program**. It takes no arguments and returns a Boolean value indicating whether or not the program was successfully loaded (true if so, false if the load process abnormally terminated).

    - `void simplesim::execute_program()`

      This member function executes an SML program in the Simplesim's memory, as described under **7.3. Execute SML Program**. It takes no arguments and returns nothing.

      Text Text

    - `void simplesim::dump_program() const`

      This member function dumps the contents of the Simplesim's registers and memory, as described under **7.4. Dump Simplesim**. It takes no arguments and returns nothing. Since this method does not modify any any of the data of the `simplesim` object that calls it, it is declared to be `const`.

  Place the following two `#include` statements at the top of this file, following any other required `#include` statements that you have coded (`<iostream>`, `<iomanip>`, etc.):

  ```
  #include "sml.h"
  #include "simplesim.h"
  ```

  This will ensure that the code that you write for your member functions will have access to the definition of the `simplesim` class and the `#define` statements that define the Simplesim instruction set.

Note that there is no mention here of a `main()` function for the program. That function will be provided to you in a separate file (described below under **8. Files I Give You**). It will be linked together with your code for the `simplesim` class during the build process initiated by the `make` command.

7. Simulating the Simplesim

The `simplesim` class will need private data members that represent the memory and registers of the machine. You should use an array of integers (length 100) to simulate the memory and five separate integer variables to simulate each of the five Simplesim registers. You might find it easiest to name the memory and register variables as follows:

```
int memory[100];
int accumulator;
int instruction_counter;
int instruction_register;
int operation_code;
int operand;
```

You should organize your program as a sequence of four steps; initialization, load, execute, and dump. Each step is described in detail below.

7.1. Initialize Simplesim

The default constructor for the `simplesim` class simulates "turning the Simpletron on". It is a simple, but necessary, step in which all five registers are initialized with zero and all 100 words of memory are initialized with the value 4444. The value 4444 was chosen, in part, because the leftmost two digits (44) are not a valid instruction.

If you prefer, the register data members can easily be initialized to 0 when they are declared using C++11's "brace or equal" syntax. In that case, there's no need to initialize them again in the constructor

7.2. Load SML Program

In the `load_program` member function, you load the SML program into the Simplesim's memory. This requires you to read the program, one line at a time, from `stdin` and stop when you encounter -99999. All input starts with the SML program, with one instruction (word) per line, and marks the end of the SML program with -99999. Load the SML program, each instruction into a word of memory, starting at memory location 00 and proceeding continuously in memory (not skipping any memory locations) until the entire program has been loaded.

As you read each line from the input file, before placing it into the Simplesim's memory, you must verify that it is a "valid" word, i.e., one that will fit (a value between -9999 and 9999, inclusive) in a Simplesim's memory cell. If you encounter an invalid word (other than -99999 which denotes the end of the program) during the program load, your program should stop loading immediately, print the appropriate error message, and return `false`.

Also, if in the course of loading the program you run out of Simplesim's memory - i.e., an SML program that is more than 100 words - your program should stop loading immediately, print the appropriate error message, and return `false`.

If the program is successfully loaded with no errors, return `true`.

7.3. Execute SML Program

Assuming a successful SML program load, the `execute_program()` member function will be called to execute the SML program. The code for this member function is essentially a loop that executes one instruction at a time. Executing an instruction is a two step process; *instruction fetch* and *instruction execute*. The body of the member function will have a structure similar to this:

```
bool done = false;
while (!done)
{
    // instruction fetch
    . . .

    // instruction execute
    . . .

    switch (operation_code)
    {
        case READ:
            . . .

        case WRITE:
            . . .

        // More cases

        default:
            . . .
    }

    if (operation_code is not branching AND !done)
        instruction_counter ++;
}
```

Instruction fetch starts by testing the value in the `instruction_counter` register. If it contains a valid memory location (00-99), then load the `instruction_register` with that word from memory and split the `instruction_register` by placing its leftmost two digits into the `operation_code` register and its rightmost two digits into the `operand` register:

```
operation_code = instruction_register / 100;
operand = instruction_register % 100;
```

If the `instruction_counter` does not contain a valid memory location, then print the appropriate error message from the table above and return. Do not attempt to set the `instruction_register`, `operation_code`, and `operand` registers.

Assuming that you have successfully fetched an instruction, the next part of the loop executes the instruction. Recall that the instruction is now sitting in the `operation_code` register and its operand is in the `operand` register.

After executing certain instructions, you must increment the `instruction_counter` register to point to the next instruction in memory for the next fetch cycle. You should increment the `instruction_counter` after executing an instruction only under the following two conditions: if the instruction that was just executed was not one of the branching instructions and if the result of executing the last instruction did not terminate the SML program, i.e., just executed `HALT` or an abend occurred.

Executing an instruction should be implemented using a `switch` statement, switching on the value in the `operation_code` register. The following is a description of how each case should be processed. Some of these will be very simple (single lines in C++) and others will require more code. Although they may be listed as a single item in the list below, each SML instruction must be a single `case` statement within your `switch`.

- `READ`. Attempt to read a word (from `stdin`) and place it into the memory location identified by `operand`. You must test the value you read before attempting to place it into memory. If it is a "valid" word (between -9999 and 9999, inclusive), then place it into memory and print a line of output. For example, after reading the value -5 and placing it into memory, your program should print the following line:

  ```
  READ: -0005
  ```

  If the value read was not a "valid" word, then you should print the appropriate error message from the previous table, do not attempt to place the word in memory, and return. The program will then proceed directly to the dump phase.

- `WRITE`. Print the word in memory from the location identified by `operand`. For example, printing the word 10 should look exactly like this:

  ```
  +0010
  ```

- `LOAD`. Place the word in the memory location identified by `operand` into the `accumulator`.

- `STORE`. Place the `accumulator` into the memory location identified by `operand`.

- `ADD`, `SUBTRACT`, `MULTIPLY`, and `DIVIDE`. Apply the arithmetic operation using the value in the `accumulator` and the word in the memory location identified by `operand`, and if successful, leave the result in the `accumulator`. The value in the `accumulator` is always the left side operand of the operation and the value in memory is always the right side operand (this makes a difference for `SUBTRACT` and `DIVIDE`).

  After performing the operation, but before placing the result into the `accumulator`, you must check for underflow and overflow. If the result is less than -9999, then you have an underflow condition. If the result is greater than 9999, then overflow. In either case, do not attempt to place the result into the accumulator (it would not fit), print the appropriate error message from the table above, and return.

When the operation is `DIVIDE`, you must perform one additional test. Before performing the operation you must check that the value in memory (at memory location `operand`) is not zero. If it is, then do not attempt to perform the operation. Print the error message from the table above and return.

- `BRANCH`, `BRANCHNEG`, and `BRANCHZERO`. These are the branching instructions. They are the only instructions that modify the value of the `instruction_counter` register. The `BRANCH` instruction simply sets the `instruction_counter` to the value found in `operand`. The other two branch instructions test the value in the `accumulator` and set `instruction_counter` to the value in `operand` if the test passes (`accumulator` is less than zero for `BRANCHNEG` or is zero for `BRANCHZERO`). Otherwise, they simply add one to the `instruction_counter`.

- `HALT`. This is the only instruction in which the `operand` register is ignored. This simply stops the execution of the SML program, and prints the following message:

  ```
  *** Simplesim execution terminated ***
  ```

  This is the only way that the SML program reaches normal termination. After executing the `HALT` statement your function should return.

If the value in the `operation_code` register is not one of those instructions (i.e., the `default` case of your `switch` statement), then your program must print the appropriate error message and return.

7.4. Dump Simplesim

At the end of every execution of your program (normal SML termination, SML program load error, or SML execution error) you must call `dump_program()` to dump the contents of the Simplesim. This means printing the contents of all five registers and printing the contents of all 100 words of memory. It must be printed in the exact following format:

```
REGISTERS:
accumulator:            +0000
instruction_counter:    00
instruction_register:   +0000
operation_code:         00
operand:                00

MEMORY:
       0     1     2     3     4     5     6     7     8     9
 0 +4444 +4444 +4444 +4444 +4444 +4444 +4444 +4444 +4444 +4444
10 +4444 +4444 +4444 +4444 +4444 +4444 +4444 +4444 +4444 +4444
20 +4444 +4444 +4444 +4444 +4444 +4444 +4444 +4444 +4444 +4444
30 +4444 +4444 +4444 +4444 +4444 +4444 +4444 +4444 +4444 +4444
40 +4444 +4444 +4444 +4444 +4444 +4444 +4444 +4444 +4444 +4444
50 +4444 +4444 +4444 +4444 +4444 +4444 +4444 +4444 +4444 +4444
60 +4444 +4444 +4444 +4444 +4444 +4444 +4444 +4444 +4444 +4444
70 +4444 +4444 +4444 +4444 +4444 +4444 +4444 +4444 +4444 +4444
```

```
80 +4444 +4444 +4444 +4444 +4444 +4444 +4444 +4444 +4444 +4444
90 +4444 +4444 +4444 +4444 +4444 +4444 +4444 +4444 +4444 +4444
```

where tabs (`\t`) are only used between the register names and their values. It is important that you use tabs and spaces as just described when writing your program (explained below).

Formatting the output correctly will require extensive use of the manipulators in `<iostream>` and `<iomanip>`. Make sure that you are familiar with the following manipulators described in the notes on Output Formatting: `setw`, `right`, `internal`, `showpos` / `noshowpos`, and `setfill`.

8. Files I Give You

The setup script will create the directory `Assign3` under your `csci241` directory. It will copy a makefile named `makefile` to the assignment directory.

Unlike the makefiles for the previous three assignments, this makefile has only a single executable target named `simplesim`. You can build the entire project for Assignment 4 simply by typing the command `make`.

The setup process will also copy a file named `main.cpp` to your assignment directory. This is a short driver program consisting of a `main()` function that does the following:

1. Creates an object of the `simplesim` class.
2. Calls `load_program()` for that object to read and load an SML program.
3. If the call to `load_program()` returns `true`, it calls `execute_program()` for the object to execute the loaded SML program. Otherwise, it just skips to the next step.
4. Calls `dump_program()` for the `simplesim` object to dump its registers and memory.

You will also receive a collection of SML programs. They include three SML programs that work (including the two example SML programs `sum` and `max`) and a collection of SML programs intended to abend. The filename and a brief description of each SML program is in the table below.

Filename	Expected Result	Description
Abending (during program load) SML Programs:		
`toobig.sml`	`*** ABEND: pgm load: pgm too large ***`	The program is too big (more than 100 words) to fit into memory.
`bigword.sml`	`*** ABEND: pgm load: invalid word ***`	Attempts to load a word greater than 9999.
`smallword.sml`	`*** ABEND: pgm load: invalid word ***`	Attempts to load

		a word less than -9999.

Abending (during execution) SML Programs:

`invalop.sml`	`*** ABEND: invalid opcode ***`	Attempts to execute an invalid instruction.
`addrs.sml`	`*** ABEND: addressability error ***`	Program without a `HALT`.
`readtoobig.sml`	`*** ABEND: illegal input ***`	Attempts to `READ` value greater than 9999.
`readtoosmall.sml`	`*** ABEND: illegal input ***`	Attempts to `READ` value smaller than -9999.
`div0.sml`	`*** ABEND: attempted division by 0 ***`	Attempts to divide by 0.
`underflow.add.sml`	`*** ABEND: underflow ***`	Continuously adds until underflow.
`underflow.mult.sml`	`*** ABEND: underflow ***`	Continuously multiplies until underflow.
`underflow.sub.sml`	`*** ABEND: underflow ***`	Continuously subtracts until underflow.
`overflow.add.sml`	`*** ABEND: overflow ***`	Continuously adds until overflow.
`overflow.mult.sml`	`*** ABEND: overflow ***`	Continuously multiplies until overflow.
`overflow.sub.sml`	`*** ABEND: overflow ***`	Continuously subtracts until overflow.

Working SML Programs:

`rw.sml`	single input value	Reads and prints

		a number.
`sum.sml`	sum of two input values	Reads and sums two numbers.
`max.sml`	maximum of two input values	Reads two numbers and prints the maximum.

Finally, you will also receive an executable file, `simplesim.key`. This is the solution to the assignment. You may use the SML programs above along with the solution (`simplesim.key`) to debug your program. The output of your program must look **exactly** like the output produced by `simplesim.key`. Your program will be tested in the following manner:

```
z123456@turing:~/csci241/Assign4$ ./simplesim.key < sum.sml > sum.key
z123456@turing:~/csci241/Assign4$ ./simplesim < sum.sml > sum.out
z123456@turing:~/csci241/Assign4$ diff sum.out sum.key
z123456@turing:~/csci241/Assign4$
```

You will only receive full credit if your output exactly matches (empty `diff`) the output of `simplesim.key` for each of the SML programs listed in the table above.

9. Hints

This is a large assignment so you might want to break it down into parts. Write part of the program, test that part thoroughly convincing yourself that it works, and then move on adding to what you've done. You might want to try the following sequence.

1. Write the initialization and dump sections first. These are the easiest parts of the assignment.

   Declaring and initializing all the variables needed to simulate the Simplesim (memory and all five registers) should not take long at all. Then you should write the dump section (skipping the program load and execution parts). Since all executions of your program will end in a dump and dumping the contents of the Simplesim will be a valuable debugging tool as you write this assignment, it's a good idea to get this done correctly right away. Once you are done with these two tasks (initialization and dump) run your program (without redirecting any `stdin`). The output should be identical to the dump presented in **Section 6.4**, that is, all five registers initialized to 0 and all 100 words of memory initialized to 4444.

2. Write the program load section.

   Write this code and insert it immediately after your initialization code. Start loading the program at memory location 00 and be sure that you stop loading program when you encounter -99999. Also, be careful to check that each word of the program that you are loading is between -9999 and 9999, inclusively, and to print the appropriate error message, stop loading, and proceed to the dump if you encounter a word that is not in that range. Also, be careful to check that the program you are loading can fit into memory (less than 100 words), and if it cannot, stop immediately, print the appropriate error message, and proceed to the dump.

Remember that the program load phase is only interested in loading valid words. It does not check if the program that is being loaded makes any sense, will work, or even if the first word loaded is a valid SML instruction. It only cares that each word is a valid word (between -9999 and 9999) and that there is enough Simplesim memory to hold the entire program.

After writing this section, you should check your work with all of the SML programs provided for you. Those that are supposed to generate program load errors (`toobig.sml`, `bigword.sml`, and `smallword.sml`) should produce the appropriate error messages before you proceed to the next part. For all the other programs, your dump should show that all the registers are still zero, that the program is loaded into memory (without the -99999 and any input that might follow), and that the rest of memory is still 2424.

3. Write the program execution section.

   Insert this code immediately after the program load section. It should only get executed if there was a successful program load. This is the largest portion of the whole assignment; however, it is not conceptually difficult. This portion of the code is essentially a `while` loop with two parts inside: instruction fetch followed by instruction execution. The instruction execution is essentially a large `switch` statement in which each case implements a single SML instruction and the default case is an `*** ABEND: invalid opcode ***`. Even this part should be done in parts itself.

   - First write just enough code to execute the `rw.sml` program. This will require you to write the loop, the instruction fetch portion, and just some of the SML instructions (`READ`, `WRITE`, and `HALT`) in the `switch` statement. Your `switch` statement must also include the default case. Save the other SML instructions for later. Before proceeding, you should also test your code using `invalop.sml` and `addrs.sml`, `readtoobig.sml`, and `readtoosmall.sml`. Make sure that you get the appropriate error messages before moving on.

   - Next write just enough code to execute the `sum.sml` program. This will require you to add some more instructions SML instructions (`LOAD`, `STORE`, and `ADD`) to your `switch` statement. Be sure that you check for overflow and underflow in your `ADD` instruction. If the result of the `ADD` instruction is less than -9999 (underflow) or greater than 9999 (overflow), do not place the result into the accumulator, print the appropriate error message, stop executing the SML program, and proceed to the dump section. Once you have done this, you can test your program using `sum.sml`.

   - Add the `DIVIDE` instruction. There is no need to check for underflow or overflow as they are not possible with `DIVIDE`. You should, however, check for division by zero.

Check your work with `div0.sml` before proceeding. Your program should print the appropriate error message.

  - Add the remaining arithmetic instructions (`MULTIPLY` and `SUBTRACT`) and the single branching instruction `BRANCH`. You will find that the `BRANCH` instruction is very simple, requiring only a single line of C++ code. Be sure to check for overflow and underflow conditions (as you did when you implemented the `ADD` instruction in the earlier step) when implementing `MULTIPLY` and `SUBTRACT`. Test your work with the all six underflow/overflow SML programs. Make sure that your program produces the correct abend message and dump for each before proceeding.

  - Finally, add the remaining branch instructions (`BRANCHNEG` and `BRANCHZERO`) and test your program using `max.sml`.

A common "trap" that many students fall into when writing assignments like this is to worry about what the SML program is doing, e.g., is the SML program going to execute an invalid instruction, is it going to enter an infinite loop, is it going to branch to the "data" portion of the program and start executing there, etc. My best advice to you is do not care about what the SML program is doing.

Your job in executing the SML program job is to simulate each instruction, one at a time. Taking each instruction one at a time is really simple. Do not care what instructions were executed before, what instructions are to come afterwards. Do not care whether or not the SML program has accidentally branched into data, or has started an an infinite loop, or if it accidentally wrote data into the program part of the program that you will eventually execute (self-modifying code), etc. Do not care about any of this stuff. **REAL** computers don't care that much about **YOUR** programs, so why should your computer care about SML programs? You shouldn't.

When executing the SML program, take each instruction one at a time, execute it to the best of your ability, and move on. If you encounter something that causes your Simplesim to abnormally terminate, then print the appropriate error message and dump. If you encounter a 1400 (`HALT`), then print the nomal termination message and dump. That's it.