CST8234 – C Programming

Assignment 01: **SMS** Simple Language Simulator

You are to write a small program to simulate the operation of a very simple computer. All the information will be handle in terms of `words`. A `word` is a simple five-digit decimal number such as +34560, -23450, 00000.

You simple computer is equipped with memory and six registers to execute code.

Memory
Your computer has 1000-word memory and these `words` are referenced by their location numbers 00000, 00001, ... 0999. Each word in memory (always a single signed five-digit decimal number) may be interpreted as an instruction to be executed, a data value, or may be uninitialized.

Accumulator
Holds 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.

Instruction Counter
Holds a memory location, a three digit number, 000, 001, ... 999. The instruction counter is used to hold the memory location of the *next* instruction to be executed.

Instruction Register
Holds a single `word`. Use to hold the instruction (a word that was pulled out of memory) that is currently being executed.

Operation Code – Operand
The operation Code holds a two digit decimal number while the operand holds a three digit decimal number. The operation code and operand registers are used to split the instruction Register, with the 2 leftmost digits and sign of instruction Register going into the operation code and the 3 rightmost digits going into the operand. For example, if the instruction register had +10009, the operation code would have +10 and the operand would have 009. Likewise, if the instruction register had -12001, the operation code would have -12 and the operand have 001.

Valid Instructions
Counts the number of valid instructions that have been executed. This register is initialized to zero during initialization and is incremented by one for each valid instruction that is executed.

Simple Machine Language (SML)
Each instruction written in SML occupies one word in memory. The 2 leftmost digits of each SML instruction are the *operation code*, which specifies the operation to be performed. The 3 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 Toke	Operation Code	Meaning
Input / Output		
READ	10	Read a word into a specific memory location
SAVE	11	Write a word into a specific memory location
Load / Store Operations		
LOAD	20	Load a word from a specific memory location into the accumulator
STORE	21	Store the word in the accumulator to a specific memory location

Arithmetic Operations		
ADD	30	Add a word in a specific memory location to the word in the accumulator (leave the result in the accumulator)
SUBSTRACT	31	Subtract a word in a specific memory location to the word in the accumulator (leave the result in the accumulator)
DIVIDE	32	Divide a word in a specific memory location to the word in the accumulator (leave the result in the accumulator)
MULTIPLY	33	Multiply a word in a specific memory location to the word in the accumulator (leave the result in the accumulator)
Transfer of control Operations		
BRANCH	40	Branch to a specific memory location
BRANCHNEG	41	Branch to a specific memory location if the accumulator is negative
BRANCHZERO	42	Branch to a specific memory location if the accumulator is zero
HALT	43	The program has completed its task

Example
Consider the following SML program which reads two numbers and computes and prints their sum.

Memory Location	Word	Instruction
000	+10007	Read A
001	+10008	Read B
002	+20007	Load A
003	+30008	Add B
004	+21009	Store C
005	+11009	Write C
006	+43000	Halt

Execution always begins at memory location 000. The word at memory location 000 (+10007) is read and interpreted as an instruction. The leftmost two digits of the word (10) represent the instruction and the rightmost three digits (007) represent the instruction's operand. The first instruction is a `READ` operation. This reads a single word from the input file and stores it in the memory location defined by the operand, in this case memory location 007. `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 001.

The next instruction (+10008) reads a second word from the input file and stores it in memory location 008. The next instruction (+20007) is a `LOAD` operation with operand 007. It takes the word found at memory location 007 (the operand) and places it into the accumulator. *All* `LOAD` *and* `STORE` *operations move data in and out of the accumulator.*
The next instruction (+30008) is an `ADD` instruction with operand 008. *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 008 (the operand), adds it to the value in the accumulator, and leaves the sum in the accumulator.

The next instruction (+21009) 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 009. Then +11009, a `WRITE` instruction, prints the word found in memory location 009, which, again, is the sum of the two input values. Finally instruction +43000, the `HALT` instruction is executed, which simply terminates the SML program (operand 000 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 007-009) were executed, however, they were important in the computation. Those words were used to store the program's variables and temporary results.

Input File

Your program will take as input a file with SML instructions, input each line into your SMS, and execute it. The input file has one instruction per line. Following the last line of the SML program will be the number `-999999` (six nines), which is not part of the SML. For example, below is the input file for the first program from the previous section. It adds -5 and 15.

```
10007
10008
20007
30008
21009
11009
43000
-999999
-5
15
```

Note that each line of the input file, other than `-999999` which is used to denote the end of program and not intended to be placed into memory, fits into a single word.

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: -00005
READ: +00015
```

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.

```
+00010
```

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

```
*********************************END EXECUTION*********************************
```

At the end of a *every* execution your program should `dump` up to and including the highest referenced address. This means dumping the contents of all six registers and the memory.

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

```
root@luna:~# ./sms < sum.sml
*********************************START EXECUTION*********************************
READ:  -0005
READ:  +0015
+0010
**********************************END EXECUTION**********************************

REGISTERS:
accumulator            +00010
instructioncounter        006
instructionregister    +43000
operationcode              43
operand                   000
ValidInstructions           7

MEMORY:

   0 +10007 +10008 +20007 +30008 +21009 +11009 +43000 -0005 +0015 +0010
```

You will note that *not* all 1000 words of memory is displayed in the dump. The dump includes only those words that are up to and including the highest memory address that was referenced during execution.

One of the first things that your program will do is read the SML program into 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 1000-word memory or a line of the input file may not fit into a word. 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 SMS 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 all program information.*

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: prg load: prg too big ***	The program is too big, more than a 1000 words to fit into memory
Invalid word	*** ABEND: prg load: invalid word ***	During a program load, one of the words in the input file was too big or too small
Execution Errors		
Invalid Operation code	*** INVALID OPCODE ***	An attempt was made to execute an unrecognizable instruction
Addressability	*** ABEND: Addressability error ***	An attempt was made to fetch an instruction from an invalid memory location

Divide by 0	*** ABEND: attempted division by 0 ***	Attempt to divide by 0
Underflow	*** ABEND: underflow ***	The result of an arithmetic operation is less than -99999
Overflow	*** ABEND: overflow ***	The result of an arithmetic operation is more than 99999
Illegal Input	*** ABEND: illegal input ***	During a READ instruction, an attempt was made to read a value that was either less than -99999 or more than 99999

SMS Implementation
You should organize your program as a sequence of four steps:

Initialization
This simulates turning on your computer. In this step, your program initializes all six registers with zero and *all* 1000 words of memory with the value 50505. The value 50505 was chosen, in part, because the leftmost 2 digits (50) is not a valid instruction.

Load a SML Program
Load the SML program into memory. This requires you to read the program, one line at a time, from `stdin` and stopping when you encounter -999999. *All* input start with the SML program, one instruction (word) per line, and mark the end of the SML program with -999999. Load the SML program, each instruction into a word of memory, starting at memory location 000 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 memory, you must verify that it is a valid `word`, i.e., one that will fit (between -99999 and 99999, inclusively) in a memory cell. If you encounter an invalid word during the program load, your program should stop loading immediately, print the appropriate error message, and proceed directly to the dump step without attempting to execute the SML program.

Also, if in the course of loading the program you run out of memory, i.e., an SML program that is more than 1000 words, your program should stop loading immediately, print the appropriate error message, and proceed directly to the dump step without attempting to execute the SML program.

Execute a SML Program
Assuming a successful SML program load, your program should execute the SML program. This step in your program is essentially a loop that executes one instruction at a time. Executing an instruction is a 2 step process; *instruction fetch* and *instruction execute*. The part of your program that executes the SML program will have a structure similar to this,

```
done = 0;
while ( !done )
{
    /* instruction fetch */

    /* instruction execute */
    switch (operationCode) {
        case READ:

        case WRITE:

        default:
```

```
        } /* end switch() */

        if (operationCode is not branching AND done != 1)
            instructionCounter ++;

    } /* endwhile */
```

Instruction fetch starts by testing the value in the `instructionCounter` register. If it contains a valid memory location (000-999), then load the `instructionRegister` with that word from memory and split the `instructionRegister` by placing it leftmost 2 digits into the `operationCode` register and its rightmost 3 digits into the `operand` register,

```
operationCode = instructionRegister / 1000;
      operand = instructionRegister % 1000;
```

If the `instructionCounter` does not contain a valid memory location, then print the appropriate error message from the table above and stop execution of the SML program. Do not attempt to set the `instructionRegister`, `operationCode`, and `operand` registers, and do not increment the `validInstructions` register. Proceed immediately to the dump step.

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

After executing certain instructions, you must increment the `instructionCounter` register to point to the next instruction in memory for the next fetch cycle. You should increment the `instructionCounter` 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). You should always increment the `validInstructions` register after executing a valid instruction, including a `HALT` instruction.

Executing an instruction should be implemented using a `switch` statement, switching on the value in the `operationCode` 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. 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`.

If the value in the `operationCode` 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, stop executing the SML program, and proceed directly to the dump phase.

For the explanation of this document, I have chosen variables with long names. I encourage you to use shorter names, for example instead of `operationCode` use `opCode`

Dump

At the end of *every* execution of your program (normal SML termination, SML program load error, or SML execution error) you must dump the content of the SMS. This means printing the contents of all six registers and printing all the words of memory up to and including the highest referenced word. For example, if loading a program occupies the first 72 words and during execution the program never references a word other than those first 72 words, then the memory portion of your dump should *only* show the contents of memory locations 000-071. On the other hand, if after loading a 72-word program and during execution the program referenced word 097 and never referenced any words between 072-096, then the memory portion of your dump should show the contents of memory locations 000-097.