ASSIGNMENT 5

Build Your Own Computer!

Due 1:15 p.m. Oct 21, 2013

Before we delve into IA-32 programming, let us "peel open" a computer and look at its internal structure. In this assignment you will be introduced machine-language programming and asked to write several machine-language programs. To make this an especially valuable experience, you will be asked to build a computer (through the technique of software-based *simulation*) on which you can execute your machine-language programs.

Part I [Machine-Language Programming] In Part II of this assignment you will be asked to create a computer called the VSM (Very Simple Machine). The VSM runs programs written in the only language it directly understands—that is, VSM Language, or VSML for short.

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

Before running a VSML program, you must *load* or place the program into memory. The first instruction (or statement) of every VSML program is always placed in location 00.

Each instruction written in VSML occupies one word of the VSM's memory (and hence instructions are signed four-digit decimal numbers). Assume that the sign of a VSML instruction is always plus, but the sign of a data word may be either plus or minus. Each location in the VSM's memory may contain either an instruction, a data value used by a program or an unused (and hence undefined) area of memory. The first two digits of each VSML instruction are the *operation code,* which specifies the operation to be performed. VSML operation codes are summarized below:

Operation Code	Meaning
Input/output operations:	
`#define READ 10`	Read a word from the terminal into a specific location in memory.
`#define WRITE 11`	Write a word from a specific location in memory to the terminal.
Load/store operations:	
`#define LOAD 20`	Load a word from a specific location in memory into the accumulator.
`#define STORE 21`	Store a word from the accumulator into a specific location in memory.
Arithmetic operations:	
`#define ADD 30`	Add a word from a specific location in memory to the word in the accumulator (leave result in accumulator).
`#define SUBTRACT 31`	Subtract a word from a specific location in memory from the word in the accumulator (leave result in accumulator).
`#define DIVIDE 32`	Divide a word from a specific location in memory into the word in the accumulator (leave result in accumulator).
`#define 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:	
`#define BRANCH 40`	Branch to a specific location in memory.
`#define BRANCHNEG 41`	Branch to a specific location in memory if the accumulator is negative.
`#define BRANCHZERO 42`	Branch to a specific location in memory if the accumulator is zero.
`#define HALT 43`	Halt—i.e., the program has completed its task. This instruction writes to the terminal the "`*** VSM execution terminated ***`" message.

The last two digits of a VSML instruction are the *operand,* which is the address of the memory location containing the word to which the operation applies. Now consider a couple of simple VSML programs.

EXAMPLE 1: The following VSML program reads two numbers from the keyboard, and computes and prints their sum.

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 instruction +1007 reads the first number from the keyboard and places it into location 07 (which has been initialized to zero). Then +1008 reads the next number into location 08. The *load* instruction, +2007, puts the first number into the accumulator, and the *add* instruction, +3008, adds the second number to the number in the accumulator. All VSML arithmetic instructions leave their results in the accumulator. The *store* instruction, +2109, places 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.

VSML programs should contain instructions for reserving memory space for the variables used in the program. For example, this program reserves three memory locations (07, 08, and 09) with the instruction +0000.

EXAMPLE 2: The following VSML program 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.

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*
08	`+4300`	*Halt*
09	`+0000`	*Variable A*
10	`+0000`	*Variable B*

TO DO: Now write VSML programs to accomplish each of the following tasks:

a. Read a series of positive integers and compute and print their sum. The end of input will be signaled by entering a negative number. (**sum.vsml**)
b. Read seven numbers, some positive and some negative, and compute and print their average. (**average.vsml**)
c. Read a series of numbers and determine and print the largest number. The first number read indicates how many numbers should be processed. (**max.vsml**)

Part II. [A Computer Simulator] In this part you are going to build your own computer. You won't be soldering components together. Rather, you will use the powerful technique of *software-based simulation* to create a *software model* of the VSM. Your VSM simulator will turn the computer you are using into a VSM, and you will actually be able to run, test and debug the VSML programs.

When you start your VSM simulator, it should begin by printing:

```
*** Welcome to VSM! ***
*** 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 -9999 to stop entering ***
*** your program. ***
```

Now assume that the simulator is running, and if we enter the program in Example 2 of Part I:

```
00 ? +1009
01 ? +1010
02 ? +2009
03 ? +3110
04 ? +4107
05 ? +1109
06 ? +4300
07 ? +1110
08 ? +4300
09 ? +0000
10 ? +0000
11 ? -9999
```

the VSM should print:

```
*** Program loading completed ***
*** Program execution begins  ***
```

Output from program

```
*** VSM execution terminated ***
```

followed by the names and contents of each register as well as the complete contents of memory. Such a printout is often called a *computer dump*. A dump after executing a VSM program would show the actual values of instructions and data values at the moment execution terminated. To help you program your dump function, a sample dump format is shown below:

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

MEMORY:
       0     1     2     3     4     5     6     7     8     9
00 +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
```

Here, **accumulator** represents the accumulator register. **instructionCounter** keeps track of the location in memory that contains the instruction being executed. **instructionRegister** contains the next instruction to be executed. You should not execute instructions directly from memory. Rather, you should transfer the next instruction to be executed from memory to **instructionRegister**. **operationCode** indicates the operation currently being performed – i.e. the left two digits of the instruction word. **operand** represents the memory location on which the current instruction operates – i.e. the rightmost two digits of the instruction currently being executed. When VSM begins execution, these variables should be initialized to zero.

TO DO: Write a C program (**vsm.c**) which will simulate the VSM. Run your VSML programs from Part I using your simulator. Test your simulator interactively as described on page 3.

You may also run your simulator in batch mode using Linux redirection. See instructions on page 5.

Your simulator should check for various types of errors. During the program loading phase, for example, each number the user types into the VSM's memory must be in the range -9999 to +9999. Your simulator should 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. 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 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, your simulator should print an error message such as:

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

and should print a full computer dump.

Part III [Extra Credit] Add the following features to your VSM simulator:

a. Extend the VSM Simulator's memory to contain 1000 memory locations to enable the VSM to handle larger programs.
b. Allow the simulator to perform remainder calculations. This requires an additional VSML instruction.
c. Allow the simulator to perform exponentiation calculations. This requires an additional VSML.
d. Modify the simulator to use hexadecimal values rather than integer values to represent VSML instructions.
e. Modify the simulator to allow output of a newline. This requires an additional VSML instruction.
f. Modify the simulator to handle string input. [*Hint:* Each VSM word can be divided into two groups, each holding a two-digit integer. Each two-digit integer represents the ASCII decimal equivalent of a character. Add a machine-language instruction that will input a string and store the string beginning at a specific VSM memory location. The first half of the word at that location will be a count of the number of characters in the string (i.e., the length of the string). Each succeeding half word contains one ASCII character expressed as two decimal digits. The machine-language instruction converts each character into its ASCII equivalent and assigns it to a half word.]
g. Modify the simulator to handle output of strings stored in the format of part (f). [*Hint:* Add a machine-language instruction that prints a string beginning at a specified VSM memory location. The first half of the word at that location is the length of the string in characters. Each succeeding half word contains one ASCII character expressed as two decimal digits. The machine-language instruction checks the length and prints the string by translating each two-digit number into its equivalent character.]

NOTE: If you decide to add any of these features to your VSM simulator, submit **extra.vsml**, and **extra.out** to demonstrate the correctness of your implementation.

Directions

1. Compile your program with:

   ```
   $ gcc –Wall -o vsm vsm.c
   ```

2. You may run your simulator in batch mode using Linux redirection. Suppose you wish to run **sum.vsml** using your simulator in batch mode. You should first make sure to comment the statement prompting the user for input. Append user input at the end of **sum.vsml**, in this case, one or more positive integers (followed by a negative number to signal the end of input). Run your program with:

   ```
   $ ./vsm < sum.vsml > sum.out
   ```

   where **sum.output** is the output from running your simulator.

3. Repeat the previous step with the other two programs from Part I.

4. When completed, upload the following files:
   - **vsm.c**
   - **sum.vsml**, **sum.out**
   - **average.vsml**, **average.out**
   - **max.vsml**, **max.out**
   - **extra.vsml**, **extra.out** (optional)

   as Assignment 5 to the course website by **1:15 p.m. Monday, October 21**. Don't forget to fully document your source!

5. Your program will be graded based on the following criteria:
   - Correctness – produces the correct result that is consistent with I/O specifications.
   - Design – employs a good modular design, function prototypes.
   - Efficiency – contains no redundant coding, efficient use of memory.
   - Style – uses meaningful names for identifiers, readable code, documentation.

6. **Do not start coding right away!** Think first about a strategy and commit your algorithm on a piece of paper. Use Stepwise Refinement Strategy as you start coding.